IWST 2014

Proceedings of the 6th edition of the International Workshop on Smalltalk Technologies

In conjunction with the 22th International Smalltalk Joint Conference
Cambridge, England, August 19, 2014

http://esug.org/Conferences/2014/IWST14

Alain Plantec
Lab-STICC, UMR 6265, University of Brest, France
alain.plantec@univ-brest.fr

Jannik Laval
Ecole des Mines de Douai, France
jannik.laval@gmail.com

Forewords

IWST, standing for International Workshop on Smalltalk Technologies, is a European Smalltalk User Group (ESUG) Conference joint event. IWST was launched in 2009, in Brest, during the 17th ESUG Conference. IWST 2014 is the sixth edition.

ESUG gathers groups of professionals and hobbyists who share an interest in the Smalltalk program- ming languages and related technologies. IWST was set up from a will to promote research activities - namely academic creative work under- taken on smalltalk use, and more generally on object technologies - apart from the ESUG main event. The goal of the workshop is to create a forum around advances or experience in Smalltalk. IWST contributes to triggering discussions and exchanges of ideas. Contributions are welcome on all aspects, theoretical as well as practical, of Smalltalk related topics such as:

- Aspect-oriented programming,
- Meta-programming,
- Frameworks,
- Interaction with other languages, • Implementation,
- New dialects or languages implemented in Smalltalk, • Tools,
- Meta-modeling,
- Design patterns,
- Experience reports

Alain Plantec and Jannik Laval

Program Committee

- Alexandre Bergel (Pleiad Lab, University of Chile, Chile)
- Noury Bouraqadi (Ecole des Mines de Douai, France)
- Damien Cassou (Lille 1, Inria, RMoD)
- Jordi Delgado (Software Department, Technical University of Catalonia (UPC))
- Marcus Denker (Inria Lille, France)
- Zoe Drey (ENSTA Bretagne - UMR 6285 Lab-STICC)
- Anne Etien (Lille 1, Inria, RMoD)
- Johan Fabry (DCC - Universidad de Chile - Santiago, Chile)
- Tudor Girba
- Mickaël Kerboeuf (UMR 6265 Lab-STICC, University of Brest, France)
- Loic Lagadec (ENSTA Bretagne - UMR 6285 Lab-STICC)
- Ciprian Teodorov (ENSTA Bretagne)
- Hernan Wilkinson (10Pines Partener and Professor at the University of Buenos Aires (UBA))

Contents

A bytecode set for adaptive optimizations

Clément Béra
RMOD - INRIA Lille Nord Europe
clement.bera@inria.fr

Eliot Miranda
Cadence Design Systems
eliot.miranda@gmail.com

Abstract

The Cog virtual machine features a bytecode interpreter and a baseline Just-in-time compiler. To reach the performance level of industrial quality virtual machines such as Java HotSpot, it needs to employ an adaptive inlining compiler, a tool that on the fly aggressively optimizes frequently executed portions of code. We decided to implement such a tool as a bytecode to bytecode optimizer, implemented above the virtual machine, where it can be written and developed in Smalltalk. The optimizer we plan needs to extend the operations encoded in the bytecode set and its quality heavily depends on the bytecode set quality.

The current bytecode set understood by the virtual machine is old and lacks any room to add new operations. We decided to implement a new bytecode set, which includes additional bytecodes that allow the Just-in-time compiler to generate less generic, and hence simpler and faster code sequences for frequently executed primitives. The new bytecode set includes traps for validating speculative inlining decisions and is extensible without compromising optimization opportunities. In addition, we took advantage of this work to solve limitations of the current bytecode set such as the maximum number of instance variable per class, or number of literals per method. In this paper we describe this new bytecode set. We plan to have it in production in the Cog virtual machine and its Pharo, Squeak and Newspeak clients in the coming year.

1. Introduction

The Cog virtual machine (VM) is quite efficient compared to popular language such as Python or Ruby, but is still far behind mainstream languages such as Java. This is because the VM does not have an adaptive Just-in-time (JIT) compiler, a tool that recompiles on the fly portion of code frequently executed to portion of code faster to run.

As we implement the adaptive optimizer as a bytecode to bytecode optimizer, we rely heavily on the bytecode set design. We need to adapt it to be suitable for optimizations and extend it with unsafe operations. The current bytecode set needs revising because of the lack of available bytecodes and the lack of unsafe operations (operations faster to run as they do not check any constraints) as well as the implementation of primitives, forbidding respectively to efficiently extend the bytecode and to inline primitive methods. In addition, the current bytecode set has well-known issues such as a maximum number of instructions a jump forward can encode and we took advantage of the bytecode set revamp to fix these problems, ending up with important simplification in the VM implementation.

We design a new bytecode set with the following improvements:

- It provides many available bytecodes to be easily and efficiently extendable.
- It features a set of unsafe operations for the runtime optimizer.
- It encodes the primitives in a way they can be inlined.
- It solves some well-known issues of the old bytecode set.

In this paper we describe the constraints we have to design a better bytecode set. We specify for each constraint if it applies for all the bytecode set designs or only in our case to design a bytecode set for adaptive optimizations.

After describing how the current bytecode is used in our virtual machine and Smalltalk clients, we discuss the current issues and missing features, then show how we solve the current issues. We also discuss some aspects of the new bytecode set and compare it to related work.

2. The Cog bytecode execution and memory model

Smalltalk has a runtime very similar to the Java Virtual Machine (JVM)[12], the Common Language Infrastructure (CLR)[6] and other common platform-independent object-oriented languages. To execute code, a high level compiler translates the Smalltalk source code into bytecode, a low level language. The bytecode is then executed by a virtual machine, being either interpreted or compiled down to na-

tive code through a just-in-time compiler. The virtual code is platform-independent and is encoded in bytes for compactness. Its byte encoding gives it the name bytecode.

A new memory manager, named Spur, has been recently introduced in the Cog VM[11]. All the figures and examples about memory representation we describe show the objects in the new Memory Manager format.

Two main bytecode sets are now supported in the Cog VM. One targets the Smalltalk clients, Squeak and Pharo, whereas the other one targets a research language named Newspeak. In this paper we focus on the bytecode set for Smalltalk clients, as the new bytecode set is for now exclusively for Smalltalk and would need to be adapted to be used with Newspeak.

2.1 Vocabulary

After looking at several bytecode sets and working on ours, we decided to describe bytecode operations by using three forms. Here is the definition of the three forms proposed, with examples from the new bytecode set:

- *single bytecode:* the instruction is encoded in a single byte. For example, the byte B0 means that the execution flow needs to jump forward by two instructions and the bytecode 4D means that the interpreter should push the boolean true on the stack.
- *extended bytecode:* the instruction is encoded in two bytes. The first byte defines the instruction and the second byte encodes an index relative to the execution of the instruction. For example, the byte E5 means that the interpreter should push a temporary variable on the stack, the index of the temporary variable being encoded in the next byte.
- *double extended bytecode:* the instruction is encoded in three bytes. The first byte defines the instruction and the second and third bytes encode an index relative to the execution of the instruction. For example, the byte FC means that the interpreter needs to push on the stack a variable in a remote array, the index of the remote array being encoded in the second byte and the index of the variable in the remote array is encoded in the third byte.

We call extended bytecode and double extended bytecode *argument bytecodes*, because they require extra byte(s) to encode the expected virtual machine behavior.

We always use the name bytecode to refer to the virtual code, i.e., the code understood by a virtual machine and not native code understood by a processor.

2.2 The Cog compiled method format

The bytecode, executable by the virtual machine, is saved in the heap - memory space reserved for object - in the form of compiled method[1]. A compiled method is an object encapsulating executable bytecode. In addition to the bytecode, a compiled method has, as shown in Figure 1:

- An object header (as every object in the system) to inform the virtual machine about basic properties such as its size, its class or its hash.
- A literal array that is used aside from the bytecode to fetch specific objects by the virtual machine to execute code.
- A compiled method header (specific to compiled method) to inform the virtual machine about basic properties such as its number of arguments or its number of temporaries.
- A source pointer to encode the way the IDE can get the method source code.

The format in memory of a compiled method is very specific. Common objects are word-aligned or byte-aligned on all their length. However, a compiled method is a mixed form of an array (its first fields in memory correspond to the compiled method header, encoded as a small integer, and its literals) and a byte array (its next fields correspond to its bytecode and its source pointer).

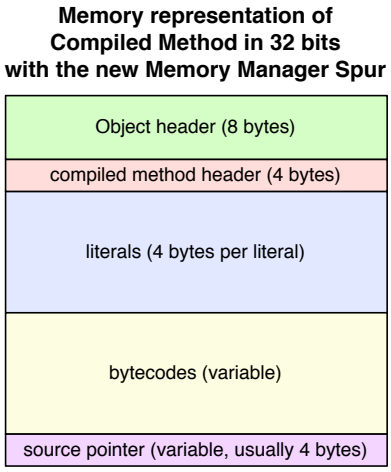


Figure 1. Memory representation of a Compiled method

Changes in the bytecode set design impacts both the memory zones for the compiled method header and for the bytecodes, but does not affect the compiled method object's header, its literal frame nor its source pointer encoding.

3. Challenges for a good bytecode set

3.1 Generic challenges

Any one who has to design a bytecode set faces some challenges. We enumerate the major ones we noticed in this subsection.

Platform-independent. Applications running on top of Cog are currently deployed on production on different operating systems: Linux, Mac OS X, Windows, iOS, Android and RISC OS and on different processors: ARM 32bits, Intel

x86 and Intel x86_64. Therefore, our bytecode set needs to be platform-independent, more specifically processor independent and operating system independent.

Compaction. To have the minimum memory footprint, a bytecode set needs to be able to encode all the compiled methods of system in the minimum number of bytes.

Easy decoding. The bytecode is mainly used by the interpreter to be executed, the JIT compiler to generate native code and in-image to analyze the compiled methods. Therefore, a good bytecode set needs to be easy to decode, decoding being used for analysis, interpretation and compilation.

Conflicts. One cannot have a very compact and very easy to decode bytecode set. Related work[9] has shown that a bytecode set can be compressed by 40% to 60% by sharing the bytecode in a Huffman table, but the bytecode becomes then harder to decode, which complicates the virtual machine interpreter, the JIT compiler and bytecode analysis. This extra difficulty also impacts performance (9% performance loss in their work). Therefore, one has to choose between easier decoding or extreme compaction. Our targets, such as the latest iPhone or the Raspberry pie have at least 256Mb of RAM. The Pharo memory footprint is usually around 20 Mb on production application. Therefore, we prefer to ease the decoding over compacting the bytecode. We want the new bytecode encoding to be at worst the same size as the old bytecode encoding but easier to decode.

Backward compatibility. Some frameworks and libraries may rely on the bytecode format to work. It includes compilers and virtual machines implementations that the designer of the new bytecode set has obviously to consider, but it also includes other frameworks such as serializers. A serializer typically reuses the bytecode encoding to serialize a compiled method. Changing the code set implies either to be backward compatible to have these tools working or fix all the frameworks and libraries.

3.2 Challenges specific to our goals

The runtime bytecode to bytecode optimizer we plan will perform classical dynamic language adaptive optimizations such as inlining based on type feeback[7] and bounds check elimination[2]. Here is a typical example:

```
MyDisplayClass>>example: anArray
    anArray do: [ :elem | self displayOnScreen: elem ].

Array(SequenceableCollection)>>do: aBlock
    1 to: self size do:
        [:index | aBlock value: (self at: index)]
```

Firstly, based on type-feed back the runtime optimizer notices that the argument of MyDisplayClass»example: is always an Array, and then inlines the message send to the array as well as the closure activation, adding a guard that triggers deoptimization if the assumption that anArray is an Array is not valid any more. We represented the guard in pseudo Smalltalk code so the code is readable, but of course a guard is typically implemented in a very efficient way in native code.

```
MyDisplayClass>>OptimizedVersion1OfExample: anArray
    Guard: [ anArray class == Array
                ifFalse: [ DynamicDeoptimization ] ].
    1 to: anArray size do: [ :index |
        self displayOnScreen: (anArray at: index) ].
```

The optimizer wants then to inline additional messages sent to anArray: size and at:. After inlining these two messages, it notices that index is always within the bounds of anArray because the to:do: selector sent on an integer enforces that the block argument is an integer between 1 and anArray size (This is typically inlined statically by the compiler). Therefore the optimizer edit the at: instruction to fetch the field of the objects at the index without checking that the index is within the bounds of anArray. In pseudo code, it would mean:

```
MyDisplayClass>>OptimizedVersion1OfExample: anArray
    Guard: [ anArray class == Array
                ifFalse: [ DynamicDeoptimization ] ].
    1 to: anArray inlinedSize do: [ :index |
        self displayOnScreen: (anArray
                inlinedNoBoundsCheckAt: index) ].
```

To be able to do this kind of optimizations, the bytecode set needs specific requirements and instructions that we detail in this subsection.

All methods should be inlinable. All methods and closure activations should be able to be inlined by the optimizer, including primitives. Inlining non primitive methods removes the overhead of a CPU call and allows the optimizer to have bigger portions of code to optimize. Inlining performance critical primitives allows the optimizer to perform additional critical optimizations such as bound check elimination. Due to inlining, optimized methods are bigger than regular methods: they may have jumps of thousands of instructions or thousands of literals. In addition, inlining a method may allow an object to access directly the instance variable of another object. Therefore, the bytecode needs to be able to encode compiled methods with:

- Inlined primitives
- Very large jump
- Very large number of literals
- Access to non receiver instance variable

Unsafe operations. Smalltalk primitives are type safe, which means that calling a primitive with inappropriate arguments will trigger a primitive failure but will not crash the execution. Therefore, each primitive needs to guarantee that the receiver and arguments have one of the expected

type before being performed. Guaranteeing the type of an object means additional CPU instructions to run. The runtime optimizer may want to encode unsafe operations, such as unchecked array access, if it can guarantee that the given primitives will not fail to avoid this type check overhead. The bytecode set needs to be extended to support a set of unsafe operations, which corresponds to inlined primitives that are optimized.

Extendable. While implementing more and more optimization passes in our optimizer, we may need to add extra unsafe operations or to encode new bytecode operations. The bytecode set needs to be easily extendable without compromising optimization opportunities and complicating the logic of the optimizer.

3.3 Current Bytecode issues

The current bytecode set could be improved compared to the generic requirements and is definitely not good enough for our requirements (specification available in Appendix A). In this subsection we describe the issues we noticed.

Argument bytecodes limits. The current bytecode set has limitations regarding the size of arguments. There is no pattern to indefinitely expand an argument bytecode. One typical example is the jump bytecode. This bytecode supports jump forward up to 1024 instructions, but no more. This is already an issue because several Pharo users ended up having a compilation error due to this problem. They had to fix it by editing their code, whereas a user should not need to understand nor see these compiler details. These issues make it impossible for some methods or closures to be inlined due to the code creating a jump too big for the encoding.

Few available bytes. One of our main concerns was the lack of available bytecodes. Only three bytecodes were available (not associated with an operation). This clearly limits the extensibility of the current bytecode set, which is required by our optimizer to easily add new instructions. We estimate that 10 available bytecodes are needed to be extended and manipulated in the future.

Primitive index implementation forbid inlining. Another issue is with the primitive index: by being encoded in the compiled method header, inlining primitives is not possible. However, the runtime optimizer needs to optimize critical primitives such as array access.

DoubleExtendedDoAnything bytecode. The old bytecode set had a bytecode named "double extended do anything" bytecode. This bytecode is able, with the proper encoding, to do almost any operation in the virtual machine. However, This bytecode creates extra difficulties in the JIT compiler to properly map the native code program counter to the bytecode program counter because you always need extra checks to know if it is a message send or something else. This bytecode does not fit with our definition of easy decoding. To be easy to decode, a bytecode set should not encode several operations in the same byte, or if this is the case then encodes very similar operation, such as push new Array and pop into array, but not operations such as a push, a store and a message send in one bytecode.

Primitive index is split. The compiled method header shown in Figure 2 has a strange primitive index field: this field is split in 2 fields, the main part of the primitive number and the high bit. This is here for backward compatibility with the 16bit Squeak version. However, as none of the Cog users has been using a Smalltalk dialects in 16bit for over multiple decades, and other recent changes have broken this compatibility. This split is now useless whereas it complicates the decoding.

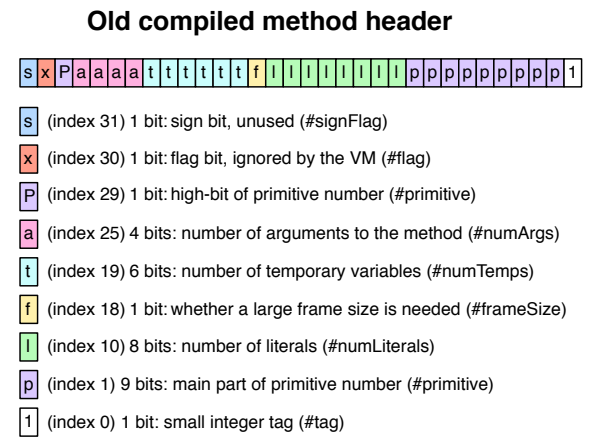


Figure 2. The old compiled method header

Immediate objects waste memory. The bytecode set misses some compact bytecode for immediate objects. For example, encoding the SmallInteger 5 requires at least 5 bytes, 4 bytes to store 5 in the literal array and 1 to 3 bytes for the push literal bytecode. It is very simple to encode and decode immediate objects in the bytecode, as their virtual machine representation is also encoded with bytes, and it greatly improves the compactness of the bytecode.

Late addition of closure bytecodes. Real closures were added in the supported Smalltalk dialects after the old bytecode set design. There was not a lot of free space in the bytecode set, so the bytecode for BlockClosure creation was added in a way that the same bytecode is used to push nil on the stack and to allocate temporary slots on stack for the BlockClosure. This makes the decoding of a BlockClosure complex for the JIT compiler and the in-image analysis due to the difficulty to fetch the number of temporaries.

Outdated constraints. The current bytecode set has a string property: most instructions are 16 bits aligned. For example, instructions 0 to 15 are mapped to push receiver

variable and instructions 16 to 31 are mapped to push temporary. We asked the old bytecode set designer, Dan Ingalls, who is also the implementor of the original Smalltalk-80. It happens that Smalltalk-80 used to run on the Xerox D, and that this 16 bits alignment was there to easily dispatch the instructions on microcoded machines. As we do not run Pharo any more on this kind of machines, we removed this constraint for the bytecode set design.

4. New bytecode set features

In this section we describe the features of the new bytecode set, starting by the ones we added for our runtime optimizer to the other one that improved the bytecode set in general. Then we explain how we convert a Smalltalk image from the old bytecode set to the new one to validate the approach.

4.1 Adaptive optimization features

Extendable instructions. One of the most notable feature of the bytecode set (specification available in Appendix B) is the addition of an extension bytecode. This bytecode, as a prefix, extends the following byte by an index encoded in a byte. In the new bytecode set, each bytecode correspond to a single instruction. It is not possible, as for the "double extended do anything" bytecode of the old bytecode set, to encode different operation in a single byte. By being a prefix, the extension bytecode does not complicate the JIT compilation, as each bytecode still represent a single instruction, and an extension bytecode just requires to fetch the appropriate instruction in a fixed distance (2 bytes further).

This feature allows for example the compiler to generate jump forward bytecode to an infinite number of instructions. The jump forward bytecode is a single extended bytecode, therefore it can jump up to 255 instructions (255 being encoded in the argument byte). By being prefixed by one extension, it can encode a jump up to 65535 instructions. If it is prefixed by two extensions, it can encode a jump up to 16777215 instructions. As one can add as many extensions as one wants, each instruction accepting extensions can be extended to the infinite.

Example: Extended conditional jump instruction
(Numbers in hexadecimal)

Byte number	EF + next byte
Name	jumpFalse:
Bit values	11101111 iiiiiiii
Explanation	Pop value on stack and jump if the value is false by a distance of distance := iiiiiiii+(ExtensionB*256)

Byte number	E1 + next byte
Name	ExtensionB (ExtB)
Bit values	11100001 bbbbbbbb
Explanation	ExtensionB ExtB := ExtB*256+bbbbbbbb

bytecode sequence	Explanation
EF 12	jumpFalse: 18
E1 05 EF 12	jumpFalse: 50D 50D = (5*FF)+12
E1 AF E1 05 EF 12	jumpFalse: ADA7BC ADA7BC = (AF*FF*FF)+(5*FF)+18

Inlined primitives and unsafe operations. The new bytecode set moved the primitive index of a method from the compiled method header to the beginning of the compiled method's bytecode (these two zones are shown in Figure 1). The primitive call is now a double extended bytecode, with the two argument bytes encoding the primitive number. The first bit of the first byte argument determines if the primitive is inlined or not. If the primitive is not marked as inlined, it automatically fails if it is not the first bytecode of the method. This change does not slow down the virtual machine due to the different method caches. Inlined primitives cannot fail, therefore one needs to be very careful while inlining a primitive to properly handle inlined primitive failure and its fall back code with control flow operations (for example, a flag on top of stack can force the execution flow to jump on a specific branch that handles the fall back code of the inlined primitive).

Byte number	F9 + next byte (> 127) + next byte
Name	callPrimitive
Bit values	11111001 0iiiiiii jjjjjjj
Explanation	call primitive at iiiiiii+(jjjjjjj*FF) fails if not the first method's bytecode On success: triggers a hard return On failure: executes fallback code

Byte number	F9 + next byte (<= 127) + next byte
Name	callInlinedPrimitive
Bit values	11111001 1iiiiiii jjjjjjj
Explanation	call inlined primitive at iiiiiii+(jjjjjjj*FF) Inlined primitives cannot fail, may be unsafe, never triggers a return, may or may not push a value on stack

To encode the primitives, we noticed that we needed to support at least 1000 primitives and 1000 inlined primitives to support all the operations we might want to implement in the next decade. We could have made CallPrimitive a single extended bytecode taking an extension according to the extendable instruction model introduced in the last paragraph, but that would complicate the VM's determination of the primitive number and the primitive error code store since the extension, being optional, would make the sequence variable length. So we decided to make it a double extended bytecode. Therefore the new representation of primitive allows 32768 primitives and 32768 inlined primitives which is more than enough.

Some of the inlined primitives are unsafe, which means that if you send them on incorrect objects you may corrupt the memory or crash the virtual machine. However, the optimizer can guarantee at compile-time that this cannot happen. The first unsafe operations we want to support are direct access to the field of an objects to optimize indexable objects access by removing bounds checks.

Access to non receiver instance variable. Instance variable access in Smalltalk does not require any specific checks, because an object cannot access any instance variable of any other objects than itself, and at compilation time the structure of the receiver is well-known. In 2008, for our efficient BlockClosure implementation, we added an extra bytecode to quickly allocate an Array on the heap and quickly access to its fields to be able to efficiently share some variables between a closure and its enclosing environment. These operations were also unchecked both for performance and because you know the size of the array at compilation time. We reused these accessing bytecodes to access the instance variables of non receiver objects.

Extendable. The new bytecode set has 15 available bytecodes. This allows us to extend it easily and efficiently.

Maximum number of literals increased. The maximum number of literal extension is quite specific. To overcome the previous limitation, we needed two changes. Firstly, we reused the free bits in the compiled method header from the primitive index to encode more literals. Secondly, we allowed the literal access bytecode to take an extension, allowing the bytecode to encode access to literals at a position over 255 in the literal frame of the compiled method.

4.2 Generic features

Overall bytecode size. In the Pharo 3.0 release (version 30848), we installed the new compiler back-end for the new bytecode set support. The system reached then 75190 compiled methods (closures are included in compiled methods). As explained in Section 2.2, the new bytecode set impacts only the compiled method header and the bytecode zone of a compiled method. However, the compiled method header has a fixed size of a word (4 bytes in 32 bits, 8 bytes in 64bits). All the bytecode zones of the compiled methods in the image used to be encoded in 2,285,892 bytes with the old bytecode set, and are now encoded in 1,960,187 bytes. The new bytecode set is therefore more compact than the old one by 14.2%. However, the difference of around 325kb is hardly noticeable on typical Pharo application that are around 20Mb.

Immediate objects compaction. Immediate objects are now encoded in the bytecode instead of in the literal array. An object in the literal array always uses a word, which corresponds to 4 bytes in 32bits. Most immediate objects can be encoded in a single byte, such as integer from 0 to 255, or the 255 most common Characters. We use that property to reduce the encoding size of most immediate objects. We added two single extended bytecode to support the encoding of SmallInteger and Character, the argument byte encoding the immediate object instead of a literal in the literal frame. These bytecodes support extension if they require extra bytes to be encoded. We have also reserved a double extended bytecode for SmallFloat, but we have not implemented it as our 64bits VM version is not stable enough. Depending on its memory manager and on its 64 bits or 32 bits form, Cog has from 1 to 3 immediate objects: SmallInteger, Character and SmallFloat. On the long term, all three will be always used, this variable number of immediate objects being due to the migration to the new Memory Manager Spur and to 64 bits.

Platform-independent. We have not introduced any platform dependent instruction, so the new bytecode set remains platform-independent.

Easy decoding. A massive improvement is related to the bytecode decoding. The bytecodes are now arranged with two simple rules:

- The bytecode are sorted by the number of bytes they need to encode their functionality: single byte bytecode are encoded between 0 and 223, extended bytecode are encoded between 224 and 248, double extended bytecode are encoded from 249 to 255.
- Within a number of byte categories, bytecodes are sorted by their functionalities: push bytecode are next to each other, send bytecodes are next to each others, ...

Easier closure decoding. A new bytecode was introduced to encode the number of temporaries in closures. We now do not need any more to walk over part of the closure bytecode to find out the number of temporaries.

About backward compatibility. Some mainstream language can hardly change their bytecode set. For example, when Java added the extra bytecode for invokeDynamic [15], the process to get it included in all virtual machines executing the Java bytecode was really tedious, and they didn't even have to edit all the bytecode compilers because this extra bytecode is provided for other languages on top of the JVM and not for Java itself. However, the Cog VM clients have two different production compilers. In addition, the most widely used serialization framework of our clients, Fuel[5], serialize the sources of a compiled method instead of its bytecode to support debugging and code edition of materialized methods in environments without decompilers. Therefore, by changing the virtual machine and the two compilers, we fixed most backward-compatibility issues.

4.3 Switching between bytecode sets to validate our approach

Difficulties with offline converters. One of the main concern, in a Smalltalk runtime, when implementing a new byte-

code set, is how to switch a snapshot from one bytecode set to another one. One solution is to implement an offline converter, that can translate the compiled method of an image from a bytecode set to another one. This solution has a big disadvantage: as long as the bytecode set implementation both image-side with the compiler back-end and VM-side with the interpreter and JIT front-end are not stable, the Smalltalk runtime crashes almost immediately at start-up and it is very hard to debug.

Multiple bytecode set support. When the Newspeak support was added to the Cog VM, the virtual machine was improved to support multiple bytecode set in the same runtime. In some Smalltalk virtual machines, such as the one of Smalltalk/X and Visual Age, the support of multiple bytecode set was implemented a while ago to be able to execute both the smalltalk bytecode and the Java bytecode. Claus Gittinger has worked on the Visual Age implementation and helped for this Cog VM improvement.

We decided to use the multiple bytecode set feature and we did not implement an offline converter. This approach has a big advantage, we can debug our bytecode compiler back end in Smalltalk on top of a VM that supports the old and the new bytecode set.

Encoding support for multiple bytecode sets. The support of multiple bytecode set is implemented part in the VM and part in the compiled method header format. The sign bit of the SmallInteger encoding the compiled method header is used to mark the method as using one or the other bytecode set, as shown in Figure 3. However, we discourage from using multiple bytecode set on top of the Cog VM for anything else than image conversion from a bytecode set to another or experiments. This is because different bytecode set have different limitations, and it may be that a method can be compiled in a bytecode set and not in another one. For example, the number of literals is limited to 255 in the old bytecode set and 65535 in the new one by assuming a CallPrimitive bytecode. This means that a method with 500 literals can be compiled n the new bytecode set but not in the old one.

Validation. Our bytecode set was validated by the implementation of a compiler back-end to generate the new bytecode out of the source code, the implementation of the interpreter front-end and the implementation of the JIT compile front-end. The whole infrastructure is running with the new bytecode set with similar performance, validating the design. This was easy as the compiler, the interpreter and the JIT compiler were designed with an abstraction layer over the bytecode set to easily change it. Moving an image from one bytecode set to another one was also not very difficult with the multi-bytecode set support feature of the VM.

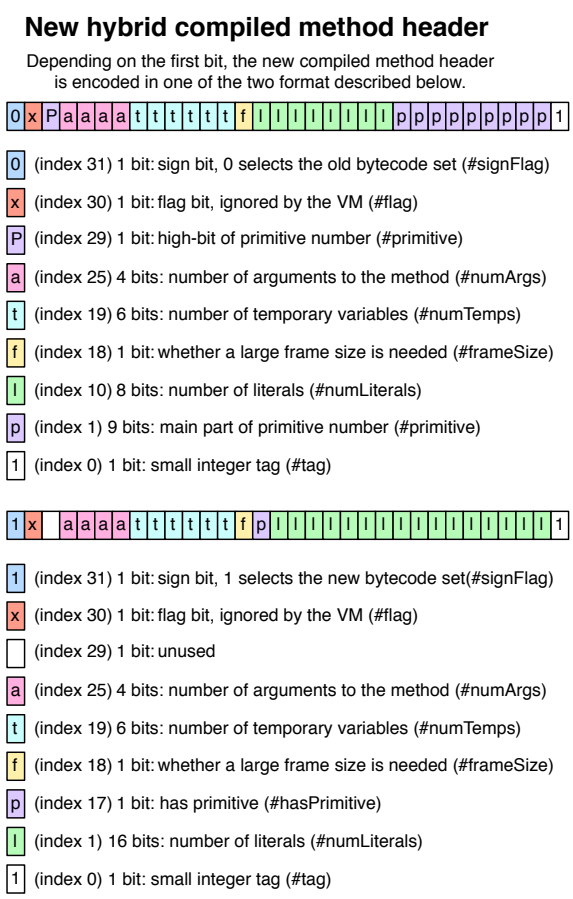


Figure 3. The new hybrid compiled method header

5. Discussion

5.1 Compaction of message sends

To improve the compactness of the bytecode set, some Smalltalk dialects as Visual Works have a specific bytecode for common message send. This bytecode is a single extended bytecode, the argument byte being the index of the common selector in a common selector array. This way, sends with common selectors are most of the time encoded in 2 bytes instead of 5 bytes because the selector is not in the literal frame, at the cost of an indirection array.

A similar feature is present in the old and new bytecode set. A list of 16 arithmetic selectors and 16 common selector have a quick encoding form. Our representation could have been extended to have this single extended bytecode, so we would have many more selectors encoded in a compact way. However, this approach has issues. For instance, common selector are not always the same. Therefore, on a regular basis, the team maintaining the programming language needs to update this common selector array depend-

ing on the new common selectors. This is problematic for some tools, such as some serializer which needs to version the serialized methods to know what common selector array it needs to use for serialization and materialization. We kept the exact same compact selectors from the old bytecode set to the new one to avoid this kind of issue. In addition, we had already reached our compaction goals, so we didn't need to add extra complexity for more compaction.

5.2 Register-based bytecode set

Our bytecode set is stack based. This design date from the time where there were both register based and stack based CPU. However, modern common CPU are now all register-based, so one can wonder if a register-based bytecode set may not be better. At Google, the Dalvik VM team chose to rely on a register-based bytecode for their Android applications[4].

We didn't move to a register-based bytecode for different reasons. The main reason is that to generate register-based bytecodes, you need to give to the compiler the number of available registers. This generates extra complexity when running the application on different platforms which may have a different number of general purpose registers. For example, on our targets, intel x86 has 8 general purpose registers whereas intel x86_64 has 16 general purpose registers. Moving from one architecture to another one requires to loose performance by using less registers than available, to recompile all the code base with the new number of general purpose registers or to have a non trivial mapping from a limited number of register to an extended one in the JIT compiler.

5.3 Threaded FFI

The Cog VM now starts to support a threaded foreign function interface (FFI) to be able to call C function without blocking the virtual machine execution. The current process implementation requires the user to fetch a global variable and to execute a message send to access the current process. This implementation was good enough but does not scale for the new multithreaded FFI. Therefore, we are considering an extra bytecode to have a direct access on the active process.

We are not sure exactly if this bytecode will be used and how. The original idea was to extend the language with an extra reserved keyword, *thisProcess*, which will push on the stack the active process the same way the active context is pushed with the reserved keyword *thisContext*. But adding an extra reserved keyword adds extra constraints to the language, so we need to study other solutions.

6. Related Works

6.1 Bytecode and primitive operations

The main difference between most bytecode sets and a Smalltalk bytecode set is that we do not encode any primitive operations in the bytecode. For example, addition is implemented as the message send named "+", and can be sent to any Object in the system. Integer addition will be performed only if the receiver is a SmallInteger. However, there are no integer addition encoded in the bytecode set that requires the operands to be SmallIntegers. Classic bytecode sets, such as the Java one[12], encodes primitive operations. For example the operation iadd in Java expects both operands to be Integers. All Smalltalk instructions expect objects of any type.

With the new bytecode set, we added encoding for inlined primitives at bytecode level, which includes typed operations. However, this encoding will only be used only by the runtime optimizer or very specific low-level tool such as an ABI compiler, and is not present by default in the Smalltalk semantics.

6.2 Bytecode set with superoperators

Several teams designed a bytecode set with superoperators to optimize the interpreter speed[3, 8, 13]. Superoperators are virtual machine operations automatically synthesized from smaller operations to avoid costly per-operation overheads. This is typically done statically at compile-time. The compiler detects common bytecode patterns and then extends the bytecode set with superoperators performing the common patterns. For an interpreter, this technique drastically improves the performance by reducing the overhead of bytecode fetching. However, in our case we would need to adapt many in-image tools, as well as the interpreter and the JIT to support it. In addition, this optimization speeds up only the interpreter, which is in most case not performance critical as our VM heavily relies on the JIT for performance. So we concluded that this optimization would cost too much time to implement compared to its benefits.

6.3 Bytecode extensions

To add infinite argument values to our argument bytecodes, we added the extension prefix bytecodes in the new bytecode set as explained in Section 4.1. We designed the extension this way to be able to encode one instruction per bytecode (only the argument is variable) and because we needed a limited number of different instructions.

Other systems have needed many more instructions, typically more than 255 which is the maximum number of different instructions you can encode in a byte. An example is the Z80 CPU bytecode[16] which needed many graphical instructions (the Z80 is the processor of the GameBoy and the SuperNintendo). In this case, they decided to use bytecode prefix to indicate to the processor to fetch the next bytecode in another bytecode table encoding other instructions. This is a convenient trick when you want an important number of instructions, but as we have much less different instructions that 255 in our virtual machine, we felt the extra complexity of this encoding didn't worth it.

6.4 Visual Works bytecode set

The Cog virtual machine is very different from the VisualWorks VM. However, they both run Smalltalk runtime, so the comparison is interesting.

BlockClosure bytecodes. One difference is that VisualWorks has another model to activate BlockClosure. When a BlockClosure is activated, it is present in the receiver slot of the context, because the BlockClosure received the block activation message. We instead have two fields in a context, one for the method activation's receiver and one for the closure. VisualWorks' model seems very pure and cleaner for the user. However, in term of the bytecode set, it means that accessing the receiver or the receiver fields in method has to be encoded differently between a method and a BlockClosure, because in one case it access the receiver of the active context whereas in the other case it access a variable from the lexically enclosed environment. In our implementation, the receiver slot of a BlockClosure context has the receiver of the homeContext, therefore accessing the receiver and the receiver's fields is the same whichever activation you are. Both implementation has pros and cons. We considered the other approach, but we preferred to simplify the virtual machine implementation at the cost of complicating a bit the model for the end user.

Loop encoding. Another difference is the support in the VisualWorks bytecode set to encode the beginning of a loop. This is convenient to be able in the JIT to generate native code in a single pass. However, loops are not very frequent in Smalltalk dialects: 5% of the compiled methods have a loop in the Pharo 3.0 release image and adding in our JIT implementation some code to handle loops didn't increase a lot the JIT complexity, therefore we preferred no to introduce this bytecode.

Common selector array. VisualWorks features a common selector array as it is described in Section 5.1. We didn't choose to add that in the Cog VM as explained in this subsection.

Non immediate entries in machine code. VisualWorks has support for non-immediate entries in machine code methods. This means that if you can guarantee that an object is not an immediate object, i.e., not a Character, a SmallInteger nor a SmallFloat, you can speed up monomorphic sends sites by targeting the method after the immediate entry. This is encoded in the bytecode with a specific send targeting the non-immediate entry in the machine code. This implementation leads to lots of complication, such as if the user change a temporary variable in the debugger leading to an immediate object being sent a message send with a non immediate entry, as well as VM-side complication as we needed to add an extra entry in the native methods, aside from the class check entry and unchecked entry.

7. Future work and Conclusion

The reason why we needed a new bytecode set was to be able to implement a runtime optimizer in the JIT compiler to optimize methods in a bytecode to bytecode fashion. This optimizer will therefore have some similarities with Soot[14], the Java bytecode to bytecode optimizer, but will be used at runtime and not statically. As the bytecode set is now ready, one needs to design and implement the runtime optimizer.

Reportedly, it is very difficult to produce correct marshaling code for FFI calls on some architectures, especially on x86_64bits[10], which is now very common. Generating marshaling code in the virtual machine is tedious, as debugging the virtual machine has always been much more complex than debugging high level languages, even with very good dedicated tools. One could implement the marshaling code of FFI calls by generating compiled methods encoding low level instructions with the unsafe operations of the new bytecode set in order to simplify this process.

In this paper, we showed how we designed a bytecode set suitable for runtime bytecode to bytecode optimizations for the Cog VM and its Smalltalk clients.

This new bytecode set encodes an extendable set of unsafe operations to provide information to the JIT compiler for producing better machine code, encodes primitives in a way the optimizer can inline them, has many available bytecodes to be easily extended and fixes the old bytecode set issues we described.

Acknowledgements

We thank Stéphane Ducasse, Stefan Marr and Marcus Denker for their reviews of early draft of this article. We thank Dan Ingalls that shared his knowledge about the old bytecode set design. We thank Claus Gittinger for helping the Cog VM to support multiple bytecode sets.

This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council, FEDER through the 'Contrat de Projets Etat Region (CPER) 2007-2013' and the MEALS Marie Curie Actions program FP7-PEOPLE-2011- IRSES MEALS.

References

[1] A. P. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland, 2009.

[2] R. Bodík, R. Gupta, and V. Sarkar. Abcd: Eliminating array bounds checks on demand. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 321–333, New York, NY, USA, 2000.

[3] P. Bonzini. Implementing a high-performance smalltalk interpreter with genbc and genvm.

[4] D. Bornstein. Dalvik virtual machine internal talk, google i/o, 2008.

[5] M. Dias, M. Martinez Peck, S. Ducasse, and G. Arévalo. Fuel: A fast general-purpose object graph serializer. *Journal of Software: Practice and Experience*, 2012.

[6] ECMA. *ECMA-334: C# Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, dec 2001.

[7] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 326–336, New York, NY, USA, 1994.

[8] A. E. Kevin Casey and D. Gregg. Optimizations for a java interpreter using instruction set enhancement. Technical report, Trinity College Dublin, sept 2005.

[9] M. Latendresse and M. Feeley. Generation of fast interpreters for huffman compressed bytecode. In *Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators*, IVME '03, pages 32–40, New York, NY, USA, 2003. ACM.

[10] A. J. Michael Matz, Jan Hubicka and M. Mitchell. System v application binary interface amd64 architecture processor supplement.

[11] E. Miranda. Cog blog. speeding up croquet and squeak with a new open-source vm from qwaq, 2008.

[12] Oracle. The java virtual machine specification, java se 8 edition.

[13] Proebsting and T. A. Optimizing an ansi c interpreter with superoperators. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 1995, pages 322–332, New York, NY, USA, 1995. ACM.

[14] P. L. Raja Vall'ee-Rai, P. P. Clark Verbrugge, and F. Qian. Soot (poster session): a java bytecode optimization and annotation framework. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (Addendum)*, OOPSLA 2000, page 113–114, New York, NY, USA, 2000. ACM.

[15] J. R. Rose. Bytecodes meet combinators: Invokedynamic on the jvm. In *Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages*, VMIL '09, pages 2:1–2:11, New York, NY, USA, 2009.

[16] ZiLOG. Z80 cpu user's manual.

A. The old bytecode set

Old Bytecode set

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Push Receiver Variable 0	Push Receiver Variable 1	Push Receiver Variable 2	Push Receiver Variable 3	Push Receiver Variable 4	Push Receiver Variable 5	Push Receiver Variable 6	Push Receiver Variable 7	Push Receiver Variable 8	Push Receiver Variable 9	Push Receiver Variable 10	Push Receiver Variable 11	Push Receiver Variable 12	Push Receiver Variable 13	Push Receiver Variable 14	Push Receiver Variable 15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Push Temp 0	Push Temp 1	Push Temp 2	Push Temp 3	Push Temp 4	Push Temp 5	Push Temp 6	Push Temp 7	Push Temp 8	Push Temp 9	Push Temp 10	Push Temp 11	Push Temp 12	Push Temp 13	Push Temp 14	Push Temp 15
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
Push Literal Constant 0	Push Literal Constant 1	Push Literal Constant 2	Push Literal Constant 3	Push Literal Constant 4	Push Literal Constant 5	Push Literal Constant 6	Push Literal Constant 7	Push Literal Constant 8	Push Literal Constant 9	Push Literal Constant 10	Push Literal Constant 11	Push Literal Constant 12	Push Literal Constant 13	Push Literal Constant 14	Push Literal Constant 15
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
Push Literal Constant 16	Push Literal Constant 17	Push Literal Constant 18	Push Literal Constant 19	Push Literal Constant 20	Push Literal Constant 21	Push Literal Constant 22	Push Literal Constant 23	Push Literal Constant 24	Push Literal Constant 25	Push Literal Constant 26	Push Literal Constant 27	Push Literal Constant 28	Push Literal Constant 29	Push Literal Constant 30	Push Literal Constant 31
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
Push Literal Variable 0	Push Literal Variable 1	Push Literal Variable 2	Push Literal Variable 3	Push Literal Variable 4	Push Literal Variable 5	Push Literal Variable 6	Push Literal Variable 7	Push Literal Variable 8	Push Literal Variable 9	Push Literal Variable 10	Push Literal Variable 11	Push Literal Variable 12	Push Literal Variable 13	Push Literal Variable 14	Push Literal Variable 15
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
Push Literal Variable 16	Push Literal Variable 17	Push Literal Variable 18	Push Literal Variable 19	Push Literal Variable 20	Push Literal Variable 21	Push Literal Variable 22	Push Literal Variable 23	Push Literal Variable 24	Push Literal Variable 25	Push Literal Variable 26	Push Literal Variable 27	Push Literal Variable 28	Push Literal Variable 29	Push Literal Variable 30	Push Literal Variable 31
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
PopInto Receiver Variable 0	PopInto Receiver Variable 1	PopInto Receiver Variable 2	PopInto Receiver Variable 3	PopInto Receiver Variable 4	PopInto Receiver Variable 5	PopInto Receiver Variable 6	PopInto Receiver Variable 7	PopInto Temp 0	PopInto Temp 1	PopInto Temp 2	PopInto Temp 3	PopInto Temp 4	PopInto Temp 5	PopInto Temp 6	PopInto Temp 7
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
Push Receiver	Push true	Push false	Push nil	Push -1	Push 0	Push 1	Push 2	Return receiver	Return true	Return false	Return nil	Return	Block Return		
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
(0) Push LongForm	(0) Store LongForm	(0) PopInto LongForm	(1) Single Extended send	(3) Double extended	Super send literal selector	(2) Second extended send	Pop Stack top	Duplicate Stack top	Push this Context	(4) Push or PopInto Array		(5) Push Temp in temp Vector	(5) Store Temp in temp Vector	(5) Pop into Temp in temp Vector	(6) Push Closure
144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
Jump 1	Jump 2	Jump 3	Jump 4	Jump 5	Jump 6	Jump 7	Jump 8	JumpFalse 1	JumpFalse 2	JumpFalse 3	JumpFalse 4	JumpFalse 5	JumpFalse 6	JumpFalse 7	JumpFalse 8
160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
Long Jump -1024 to -769	Long Jump -768 to -513	Long Jump -512 to -257	Long Jump -256 to -1	Long Jump 0 to 255	Long Jump 256 to 512	Long Jump 513 to 768	Long Jump 768 to 1024	Long JumpTrue 0 to 255	Long JumpTrue 256 to 511	Long JumpTrue 512 to 768	Long JumpTrue 769 to 1024	Long JumpFalse 0 to 255	Long JumpFalse 256 to 511	Long JumpFalse 512 to 768	Long JumpFalse 769 to 1024
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
+	-	<	>	<=	>=	=	~=	*	/	\\	@	bitShift:	//	bitAnd:	bitOr:
192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
at:	at: put:	size	next	nextPut:	atEnd	==	class	Block Copy:	value	value:	do:	new	new:	(Point) x	(Point) y
208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
Send 0 args selector at 0	Send 0 args selector at 1	Send 0 args selector at 2	Send 0 args selector at 3	Send 0 args selector at 4	Send 0 args selector at 5	Send 0 args selector at 6	Send 0 args selector at 7	Send 0 args selector at 8	Send 0 args selector at 9	Send 0 args selector at 10	Send 0 args selector at 11	Send 0 args selector at 12	Send 0 args selector at 13	Send 0 args selector at 14	Send 0 args selector at 15
224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
Send 1 args selector at 0	Send 1 args selector at 1	Send 1 args selector at 2	Send 1 args selector at 3	Send 1 args selector at 4	Send 1 args selector at 5	Send 1 args selector at 6	Send 1 args selector at 7	Send 1 args selector at 8	Send 1 args selector at 9	Send 1 args selector at 10	Send 1 args selector at 11	Send 1 args selector at 12	Send 1 args selector at 13	Send 1 args selector at 14	Send 1 args selector at 15
240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255
Send 2 args selector at 0	Send 2 args selector at 1	Send 2 args selector at 2	Send 2 args selector at 3	Send 2 args selector at 4	Send 2 args selector at 5	Send 2 args selector at 6	Send 2 args selector at 7	Send 2 args selector at 8	Send 2 args selector at 9	Send 2 args selector at 10	Send 2 args selector at 11	Send 2 args selector at 12	Send 2 args selector at 13	Send 2 args selector at 14	Send 2 args selector at 15

(0) jjkkkkkk, jj = Receiver Variable, Temporary Location, Literal Constant / Illegal, Literal Variable, kkkkkk the index
(1) 32 first lit as selector and up to 7 args
(2) 64 first lit as selector and up to 3 args
(3) iiijjjjj kkkkkkkk, iii = Send, Send Super, Push Receiver Variable, Push Literal Constant, Push Literal Variable,
Store Receiver Variable, Store-Pop Receiver Variable, Store Literal Variable, index = kkkkkkkk, for sends jjjjj = numArgs
(4) jkkkkkkk, Push (Array new: kkkkkkk) (j = 0) or Pop kkkkkkk elements into: (Array new: kkkkkkk) (j = 1)
(5) kkkkkkkk jjjjjjj, temp At kkkkkkkk In tempVect At jjjjjjj
(6) llllkkkk jjjjjjj iiiiiii, Push Closure Num Copied llll Num Args kkkk BlockSize jjjjjjjiiiiiii

Figure 4. The old bytecode set

B. The new bytecode set

New Bytecode set

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Push Receiver Variable 0	Push Receiver Variable 1	Push Receiver Variable 2	Push Receiver Variable 3	Push Receiver Variable 4	Push Receiver Variable 5	Push Receiver Variable 6	Push Receiver Variable 7	Push Receiver Variable 8	Push Receiver Variable 9	Push Receiver Variable 10	Push Receiver Variable 11	Push Receiver Variable 12	Push Receiver Variable 13	Push Receiver Variable 14	Push Receiver Variable 15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Push Literal Variable 0	Push Literal Variable 1	Push Literal Variable 2	Push Literal Variable 3	Push Literal Variable 4	Push Literal Variable 5	Push Literal Variable 6	Push Literal Variable 7	Push Literal Variable 8	Push Literal Variable 9	Push Literal Variable 10	Push Literal Variable 11	Push Literal Variable 12	Push Literal Variable 13	Push Literal Variable 14	Push Literal Variable 15
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
Push Literal Constant 0	Push Literal Constant 1	Push Literal Constant 2	Push Literal Constant 3	Push Literal Constant 4	Push Literal Constant 5	Push Literal Constant 6	Push Literal Constant 7	Push Literal Constant 8	Push Literal Constant 9	Push Literal Constant 10	Push Literal Constant 11	Push Literal Constant 12	Push Literal Constant 13	Push Literal Constant 14	Push Literal Constant 15
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
Push Literal Constant 16	Push Literal Constant 17	Push Literal Constant 18	Push Literal Constant 19	Push Literal Constant 20	Push Literal Constant 21	Push Literal Constant 22	Push Literal Constant 23	Push Literal Constant 24	Push Literal Constant 25	Push Literal Constant 26	Push Literal Constant 27	Push Literal Constant 28	Push Literal Constant 29	Push Literal Constant 30	Push Literal Constant 31
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
Push Temp 0	Push Temp 1	Push Temp 2	Push Temp 3	Push Temp 4	Push Temp 5	Push Temp 6	Push Temp 7	Push Temp 8	Push Temp 9	Push Temp 10	Push Temp 11	Push Receiver	Push true	Push false	Push nil
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
Push 0	Push 1	Push context Push process	Duplicate Stack Top					Return Receiver	Return true	Return false	Return nil	Return Top	BlockReturn Nil	BlockReturn Top	Nop
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
+	-	<	>	<=	>=	=	~=	*	/	\\	@	bitShift:	//	bitAnd:	bitOr:
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
at:	at: put:	size	next	nextPut:	atEnd	==	class		value	value:	do:	new	new:	(Point) x	(Point) y
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
Send 0 args selector at 0	Send 0 args selector at 1	Send 0 args selector at 2	Send 0 args selector at 3	Send 0 args selector at 4	Send 0 args selector at 5	Send 0 args selector at 6	Send 0 args selector at 7	Send 0 args selector at 8	Send 0 args selector at 9	Send 0 args selector at 10	Send 0 args selector at 11	Send 0 args selector at 12	Send 0 args selector at 13	Send 0 args selector at 14	Send 0 args selector at 15
144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
Send 1 args selector at 0	Send 1 args selector at 1	Send 1 args selector at 2	Send 1 args selector at 3	Send 1 args selector at 4	Send 1 args selector at 5	Send 1 args selector at 6	Send 1 args selector at 7	Send 1 args selector at 8	Send 1 args selector at 9	Send 1 args selector at 10	Send 1 args selector at 11	Send 1 args selector at 12	Send 1 args selector at 13	Send 1 args selector at 14	Send 1 args selector at 15
160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
Send 2 args selector at 0	Send 2 args selector at 1	Send 2 args selector at 2	Send 2 args selector at 3	Send 2 args selector at 4	Send 2 args selector at 5	Send 2 args selector at 6	Send 2 args selector at 7	Send 2 args selector at 8	Send 2 args selector at 9	Send 2 args selector at 10	Send 2 args selector at 11	Send 2 args selector at 12	Send 2 args selector at 13	Send 2 args selector at 14	Send 2 args selector at 15
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
Jump 1	Jump 2	Jump 3	Jump 4	Jump 5	Jump 6	Jump 7	Jump 8	JumpTrue 1	JumpTrue 2	JumpTrue 3	JumpTrue 4	JumpTrue 5	JumpTrue 6	JumpTrue 7	JumpTrue 8
192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
JumpFalse 1	JumpFalse 2	JumpFalse 3	JumpFalse 4	JumpFalse 5	JumpFalse 6	JumpFalse 7	JumpFalse 8	PopInto Receiver Variable 0	PopInto Receiver Variable 1	PopInto Receiver Variable 2	PopInto Receiver Variable 3	PopInto Receiver Variable 4	PopInto Receiver Variable 5	PopInto Receiver Variable 6	PopInto Receiver Variable 7
208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
PopInto Temp 0	PopInto Temp 1	PopInto Temp 2	PopInto Temp 3	PopInto Temp 4	PopInto Temp 5	PopInto Temp 6	PopInto Temp 7	Pop Stack Top							
224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
Extension A	Extension B	Extended Push Receiver Variable	Extended Push Literal Variable	Extended Push Literal	Extended Push Temporary Variable	Push N Closure Temps	Push Integer	Push Character	Push or PopInto Array	Extended Send Literal selector	Send To superclass Literal selector	Trap on Behavior/ Array of Behavior	Extended Jump	Extended Pop And Jump True	Extended Pop And Jump False
240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255
Extended PopInto Receiver Variable	Extended PopInto Literal Variable	Extended PopInto Temporary Variable	Extended Store Receiver Variable	Extended Store Literal Variable	Extended Store Temporary Variable				Call Primitive Call inlined Primitive	*Reserved for Push Float*	PushClosure Copy	Push Temp in temp Vector	Store Temp in temp Vector	Pop into Temp in temp Vector	

Figure 5. The new bytecode set

An extensible constraint-based type inference algorithm for object-oriented dynamic languages supporting blocks and generic types

Nicolás Passerini
Universidad Nacional de Quilmes
Universidad Nacional de San Martín
UTN – F.R. Buenos Aires
npasserini@gmail.com

Pablo Tesone
Universidad Nacional del Oeste
Universidad Nacional de Quilmes
Universidad Nacional de San Martín
tesonep@gmail.com

Stephane Ducasse
RMoD Project-Team, Inria
Lille–Nord Europe / Université de Lille
stephane.ducasse@inria.fr

Abstract

Dynamically typed languages promote flexibility and agile programming. Still, their lack of type information hampers program understanding and limits the possibilities of programming tools such as automatic refactorings, automated testing framework, and program navigation. In this paper we present an extensible constraint-based type inference algorithm for object-oriented dynamic languages, focused on providing type information which is useful for programming tools. The algorithm is able to infer types for small industrial-like programs, including advanced features like blocks and generic types. Although it is still an early version, its highly extensible and configurable structure make our solution a useful test bench for further investigation.

Categories and Subject Descriptors F-3.3 [*Logics and meanings of programs*]: Studies of Program Constructs

General Terms type inference, dynamic languages, concrete types, abstract types

Keywords dynamic languages, type inference, ide, tools, automatic refactorings, type annotations, Smalltalk, Pharo.

1. Introduction

Dynamically-typed languages have many advantages, such as a strong flexibility, reduced development time and code size [Tra09]. Still their lack of type information hampers program understanding and limits the possibilities of programming tools such as an automated testing framework [Ahm13, GKS05], program navigation and refactorings [Opd92] or smart suggestions [RL13]. In addition type information improves programmer understanding by showing method parameter type, or the messages that can safely be sent to a variable. Such properties are important when a programmer should join in an existing team with a large code base to maintain. The same difficulties appear when learning how to use a new API. Types provide an important conceptual framework for the programmer, useful for program design, maintenance and understanding [Bra04].

There are many different approaches (such as type inferencing [Suz81, GJ90, Unt12, HPHf11, OPSb92] or gradual typing [ACF+13]) to provide static type information to dynamic languages. However, industrial programming environments for object-oriented dynamically-typed languages take little or no advantage of these ideas. One of the causes for this is that frequently these approaches have somewhat limited applicability on industrial programming, because they work on a limited version of a general purpose programming language [Gar01, SS04] or even define their own language specifically for that purpose [PS91, Hen94, OPSb92]. With Gradual Typing, other approaches propose to modify the language semantics and to add type annotations to the language itself [Gra89, ST07, WB10]. In addition many solutions are partial [Suz81, PS91] and do not cover complex part of the language such as closures. Finally there are approaches which are successful for program optimisation and delivery, but its performance would be unacceptable for interactive uses such as automatic refactorings and code completion [Age96].

The programming environment chosen for this development is Pharo[1]. Pharo is an Open Source programming language and environment. It is inspired in Smalltalk and it is used in many industrial applications.

[1] http://pharo.org/

The objective of this work is to create a *pluggable type system* [Bra04] which can improve a programming environment by feeding the programming tools with type information. Our main contributions are:

- a prototype implementation of an algorithm which computes type information for an existing Pharo program, supporting some of its most complex features, like block closure and generic data types,
- a strategy for combining both *concrete* and *abstract types*,
- a modular, configurable and extensible algorithm which provides a framework for future analysis and, for example, tune it alternatively for better performance or precision.

The rest of this paper is structured as follows. In Section 2 we present the problems arising from the lack of type information in dynamic languages. Section 3 proposes an inference algorithm to get this information from an existing code base without modifying it, also explaining how the different type variables and constraints are built and how they interact. Section 4 explains how we address some advanced features: blocks and generic data types. In Section 5 we show a small example of the type information that can be obtained using our algorithm on a small but non-trivial program. Section 6 analyses the pros and cons of our solution, while Section 7 compares our proposal with other type inference alternatives. Finally, we summarize our contributions in Section 8, along with some possible lines of further work derived from this initial ideas.

2. The Challenge of Lack of Type Information

Type information can be useful to improve programming environments (IDEs). For example, an automatic method rename tool could use type information to select more precisely the message sends that should be updated when a method is renamed. Also type information can be used to show the programmer a list of messages that can be sent to a variable or what kind of objects a method expects to receive as parameter. Moreover, code navigation can be improved for example if the IDE can provide a more acurate set of the possible receivers of a message.

In several languages, the programmer is forced to provide type information, for example associating each variable with a *type annotation*. In these languages, static type information is available and IDEs can take advantage of it. *Dynamically-typed languages* such as Pharo, Javascript or Ruby allow the programmer to avoid type annotations. This simplifies prototyping and modelling, albeit sometimes limiting the power of some programming tools.

A *type inferer* is a tool that automatically computes types for some part of a program which lacks type annotations. Statically typed languages such as Haskell or ML have a large experience with type inference [Hud89]. However in those languages the type system is integrated with the language, *i.e.,* we are unable to run a program with a type error. It is a much more difficult task to apply type inference to a program written with no notion of typing, which is the case in object-oriented dynamically typed languages.

Dynamic object-oriented languages pose several difficult challenges to type inference. *Subtype polymorphism* allows a variable to hold objects of different types (and not related types). For this reason, when a message is sent to an object, it is not simple to see which method is going to be executed. Also, for container objects such as collections it is not sufficient to know its type (*e.g.,* Set or OrderedCollection), instead we need to have information about the type of its elements. This capability of an object to be instantiated using different types for its instance variables is known as *generic data types*. [CW85]

The purpose of this work is to build a type-inferer system that can be practically used to improve programming tools, such as automatic refactorings, code navigation and smart suggestions. To achieve this objective, a type-system should comply to the following:

- It should *work on existing real-world programs*, *i.e.,* we may not restrict the use of the language or create an ad-hoc language that fits the needs of our type-system. Also, we should avoid requiring the programmer to add type annotations to its program in order to use our tools.
- The type-inferer should be *responsive*, *i.e.,* it should be able to run while a user is coding, for example each time he compiles a method. An execution time of even a few seconds is already unacceptable. To fulfill this goal, we think that a type system should be able to work *incrementally*, *i.e.,* when a method is modified, rebuild type information only for that method and re-use all the information computed for the rest of the system.

On the other hand, since our goal is restricted to feed tools with type information, we do not intend to detect type errors.

A usual purpose of a type system is to detect programming errors [Mil78]. Though, dynamic language programs frequently make use of programming idioms that are be very hard or even impossible to type-check, as those described in the work of Allende *et al.,* [ACF+13], such as the use of the same local variable to hold non related objects, or any use of meta programming techniques. Having to avoid those idioms to conform to a static type system, would cut off a significant portion of the sense of a dynamic language. Therefore, dynamically typed language programmers prefer other tools to detect programming errors, such as unit testing [GNDc04, Eck03, Mar03].

For these reasons, our solution does not intend to detect errors in the program under analysis. Instead, we assume it correct and just try to infer type information to help the

programmer understand the program or modify it with more confidence.

Dynamically-typed languages do not provide type information for their core libraries. Since such libraries are just programs expressed in the same language, it is possible to infer their types with the same tools used to analyse individual programs. Only primitives, which tend to be scarce in most dynamic languages, require special inferencing handling because they often represent the connection to external world such as C libraries or plugins. However, because of their complexity and generality, the analysis of these libraries often will consume most of the time of a type-inferer. Since application programmers seldom modify core libraries, we choose to consider them also as primitives and feed the type-inferer with their types [Gar01].

Type systems can be characterized from concrete to abstract [Age96]. *Abstract types* specify an object's interface, while *concrete types* describe its implementation. Abstract types can be modelled as a set of messages an object should understands. Concrete types are usually modelled as sets of concrete classes. Most type-inference systems on object-oriented dynamic languages focus on concrete types, because they are useful for some purposes such as program optimisation and delivery. Also they are simpler to understand to programmers without a heavy background on type systems. However, concrete types cannot express the type of a method parameter whithout knowing all of its clients (*closed-world assumption*). A somewhat novel feature of our approach is to combine concrete and abstract type information.

3. Basic Algorithm

Our solution implements a type inference algorithm based in constraints generation which combines concrete and abstract types. It associates each expression (in our implementation each AST[2] node) with a *type variable*, it analyses the code gathering constraints for each type variable and associates each type variable with possible types so that all constraints are meet.

The constraint solving algorithm is divided into independent *tasks* which are related through a workflow. The algorithm is *extensible* and *configurable*, new tasks can be created and the workflow can be changed. In this way, our solution provides both a useful tool for type inference research and a configurable framework adaptable to different purposes.

The algorithm is *iterative*, *i.e.,* each task is executed multiple times and each time it may produce new information, based on the information obtained by other tasks or by a previous execution of the same task. After a task is executed, the workflow decides which task to execute next, depending on if the previous task was successful in obtaining more type information or not.

[2] Abstract Syntax Tree

Also the solution is capable of handling generic data types and inferring type information on independent parts of the code, allowing the use of unbound parameters in the root analysed method; in other words it can infer type information in a program without a main method. This is achieved by the combined use of concrete and abstract types. These two sources of information provides a better understanding of the analysed programs.

The input of our solution is one or more initial methods, as it was said, these methods can have unbound parameters. The algorithm will automatically select which other methods have to be analysed. The answer of our algorithm is a set of restrictions gathered into type variables, which are associated to each expression in the program. The restrictions specify the possible types of the values of the expressions, using both concrete and abstract types. We name the set of known classes that comply with those restrictions the *result* of a type variable.

Our solution allows one to manually specify the types of some methods, thus avoiding to infer their type information. In fact a few of those *type specifications* are necessary for the algorithm to work. This is the case for

- Virtual machine primitives
- Methods of generic classes, such as Collections (*cf.* Section 4.2)

However, type specifications are also useful to set boundaries to type inference. For example, an application developer could take advantage of having type specifications for the language core and other libraries he uses, as this would shorten inference time. Also type specifications can improve precision in cases where the algorithm cannot infer the most precise type.

3.1 Type Variables and Constraints

As it was said previously, each expression has a type variable associated with it. For example, for the expression: `z := x addTo: y`, the following type variables are generated: t_x, t_y, t_z and $t_{(x\ addTo:\ y)}$. For each variable in the code, we generate a single type variable which is shared among all occurences of the variable. Pharo has several types of variables including class variables, instance variables, local variables, method parameters and block parameters. Also a type variable is created for the return type of each method.

The different type variables are related by constraints. There are only two types of constraints: *subtype* constraints and *receive message* constraints.

3.1.1 Subtype constraints.

A subtype constraint "t_x is subtype of t_y", written $t_x \prec t_y$ establishes that

1. If we know that `C` is a valid result for the subtype t_x, then `C` should be also a valid result for the supertype t_y.

2. If we know that the result of the supertype t_x has to understand message #m1, then the result of the subtype t_y also has to understand #m1

Therefore, these constraints can be used to propagate type information. Each time we obtain some information about a type variable, we can use it to deduce something about its subtypes or supertypes.

It is worth mentioning that subtyping is a relationship between type variables and it does not have a direct relationship to inheritance. Subtype constraints are generated in the following cases:

- **Assignment:** a := b implies that $t_b \prec t_a$.
- **Return:** If we find the expression ^a in the method #m of class C, we can deduce $t_a \prec t_{(C \gg \#m.\uparrow)}$, where $t_{(C \gg \#m.\uparrow)}$ is the type variable associated to the return type of the method.
- **Parameter passing** From the expression obj m: a and for each class C that is a valid result for t_{obj} we can deduce that $t_a \prec t_{(C \gg \#m:.1)}$, where $t_{(C \gg \#m:.1)}$ is the type variable associated to the first formal parameter of the method #m: in the class C

The first two cases occur during the analysis of the AST (*cf.* Section 3.3), while the last one is a bit more complex and therefore it can only derived by the constraint solving tasks (*cf.* Section 3.3).

3.1.2 Receive message constraint

A receive message constraint establishes that any valid result of a type variable has to understand the specified message. Moreover, it establishes relationships to other type variables which will be related to the arguments and the result type of the messages. For example, from the expression x addTo: y we deduce:

t_x *should understand* #addTo: *with type* $t_y \rightarrow t_{(x \text{ addTo: } y)}$

This means not only restricting the possible results for t_x but also associating the variable t_y to the argument of the message and the variable $t_{(x \text{ addTo: } y)}$ to its return type. This associations are not useful at this point, but they will become relevant once we assign t_x to a concrete type (or many) and hence #addTo: to a specific method (or many).

3.2 Type Information

All the type information collected by the algorithm is related to a type variable. For each type variable (and hence for each AST Node) we collect:

- The set with all the type variables which are direct *subtypes* of this variable.
- The set with all the type variables which are direct *supertypes* of this variable.
- The set of *messages* that the types assigned to this type variable must understand *msgs(t_x)*.
- The *minimal set of concrete types* that must fit in the type variable: *min(t_x)*.
- The *maximal set of concrete types* that could fit in the type variable: *max(t_x)*.

Subtypes and supertypes express relationships between type variables and are created by the constraint generation task (*cf.* Section 3.1). They will allow to propagate type information between type variables during constraint solving. For example if we know that $t_x \prec t_y$ and t_x can be of type SmallInteger, then t_y must also be able to hold SmallInteger's. Following the relationship in the opposite way, if t_y must understand #addTo:, then its subtype t_x must also understand it.

The set of messages is also created by the constraint generation task. Also, each message send is related to other type variables representing the arguments and the return value of this message send.

The minimal set of concrete types is a set of Pharo classes. A class C is included in *min(t_x)* each time the algorithm can find evidence that an instance of C can be the result of evaluating the AST Node related to t_x. Examples of evidence are:

- *Literal values.* For example from the expression x := 37 follows that SmallInteger $\in$ *min(t_x)*.
- *Primitive return types.* For example from x := OrderedCollection basicNew follows that OrderedCollection $\in$ *min(t_x)*.
- *Propagation due to subtype/supertype relationships.* For example, from this sequence of assignments: x := 37. y := x, we know that SmallInteger $\in$ *min(t_x)* (from the first assignment) and $t_x \prec t_y$ (from the second one). Then we can deduce that SmallInteger $\in t_y$

The maximal set of concrete types of a type variable t_x is the set of all Pharo classes that could possibly be the result of evaluating the AST node related to t_x. If a class C is not included in *max(t_x)*, it is because we have evidence that the related AST node could never be evaluated to an instance of C without producing an error. There are three ways of compute *max(t_x)*:

- *Primitive parameter types.* Primitives usually can accept only a limited number of classes as parameters, or even a single one.
- *Message sends.* If *msgs(t_x)* = {#add:, #remove: } then *max(t_x)* is the set of all classes that understand both #add: and #remove:.
- *Propagation due to subtype/supertype relationships*. For example, $t_y \prec t_x$ and *max(t_x)* = {A, B }, then *max(t_y)* $\subseteq$ {A, B }.

In our algorithm the minimal set starts empty and is enlarged as the algorithm finds new evidence. On the other

hand, the maximal set starts containing all the classes in the image[3] and it is reduced as the algorithm progresses. During type inference always holds that $min(t_x) \subseteq max(t_x)$. Since $min(t_x)$ can only grow and $max(t_x)$ can only shrink, the final type computed will be between those two limits.

3.3 Tasks

Our algorithm is divided in 6 independent tasks:

- Generate type variables and constraints for a method.
- Propagate minimal set of concrete types.
- Link a message-send with a method.
- Propagate maximal types.
- Compute max concrete types from message sends.
- Infer a minimal type from a singleton maximal type.

The following sections explain each of these tasks.

Constraint generation. This task has the responsibility of creating the type variables for a method. For each method that has to be analysed, it can proceed in two ways. If there is a type specification for that method, it generates only type variables for its formal parameters and return type, with fixed types. If a type is not present (which is the general case), the task walks through the AST of the method generating a type variable for each expression in the method. In this case, while traversing the AST, it also collects the subtype relationships between type variables, and each message sent to an expression is translated intro a constraint applied to the associated type variable, as described in Section 3.1.

This is the starting point of the algorithm and the only mandatory task. Since the input to the algorithm is a set of methods, the first execution of this will compute type variables and constraints for each of these input methods. As other tasks are executed, the algorithm will discover other methods that have to be analysed. These methods will not be analysed immediately but enqueued. The workflow has the resposibility of deciding when to re-execute this task.

Propagate Minimal Set of Concrete Types. The main objective of this task is the propagation of the minimal types detected in previous tasks. For each pair of type variables t_x and t_y such that $t_x \prec t_y$, we have to update $min(t_y)$ so that:

$$min(t_y) := min(t_y) \cup min(t_x)$$

i.e., the minimal type set of the supertype t_y is enlarged to include the minimal type set of the subtype t_x.

For efficiency reasons, the algorithm keeps track of which concrete types have already been propagated. By doing so, in each execution of the task we also are able to tell if new information has been discovered or not, which is necessary for the workflow to decide which task to run next.

[3] This is only from a theoretical poin of view. For efficiency reasons, in the actual implementation the set of all classes is never computed.

Link a message sent with a method. The main objective of this task is to link the type variables associated to a message-send with the type information associated to a specific method. To be able to do this, we have to infer the possible method that could be executed as response to the message send. Those methods are looked up in the classes from the minimal type set of the receiver. Thus, for each message-send x m: y and for each $C \in min(t_x)$, we look up the type variables for the formal parameters and return type of the method C≫#m and create the following constraints:

$$t_y \prec t_{(C\gg\#m.1)} \qquad\qquad t_{(C\gg\#m.\uparrow)} \prec t_{(x\ m:\ y)}$$

The use of type variables allows us to create these subtype relationships without having actual type information for the method C≫#m. If the type information is not available, we only create the type variables and enqueue the method for being processed by the constraint generation task (*cf.* Section 3.3).

As in the previous task, for each message-send, the algorithm keeps track of the concrete types of the receiver for which we already generated constraints, avoiding to process the same method twice for the same message-send and allowing the task to inform the workflow if it has make some progress.

Propagate Minimal Set of Concrete Types. This task propagates the information of the maximal types set is propagated through in a similar way as explained in Section 3.3. Still, there are two big differences in the operation of both tasks.

First, the nature of the maximal types set mandates to propagate the information in the opposite direction, *i.e.,* from supertypes to subtypes. If $t_x \prec t_y$ this means that the final result of t_x has to be included in the final result of t_y. Since $max(t_x)$ is an upper limit of the final result of t_x, it also works as un upper limit t_y.

Second, as we said before, the initial maximal types set contains all the classes in the image and it shrinks as we obtain more information. Therefore, the propagation process leaves only the concrete types present in both the type variable and its supertype:

$$max(t_x) := max(t_x) \cap max(t_y)$$

Compute max concrete types from message sends. In this task the information of the maximal concrete types are filtered using the messages sent to the type variable. Only the concrete types which implement all the messages are kept in the maximal set.

Infer a minimal type from a singleton maximal type. This is the only task in the current implementation which combines information from the minimal and maximal sets. When we find out that the maximal types set of a type variable has only one concrete type ($max(t_x) = \{C\}$), then the minimal types set should contain exactly the same type.

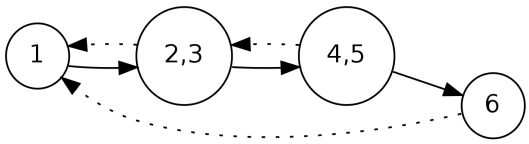
$$min(t_x) := max(t_x)$$

3.4 Coordination of tasks

The tasks of the algorithm are coordinated by a workflow. The workflow is organized in *run levels*, each run level has one or more tasks to execute. The workflow executes the run levels in a sequential way, deciding which run level to execute next depending on if the previous run level has produced new information or not.

The workflow is fully configurable, allowing to create new tasks, remove some of the current tasks and change how the tasks are organized run levels. Having a configurable workflow allows to use our algorithm as a testing bench for different configurations, comparing their performance, accuracy and fitness for different purposes. Also, different configurations could be useful for different purposes. For example, a smart suggestions tool requires a very fast type inferer, even at the cost of loosing precision, while a type inferer used for compiler optimization can be allowed to run for hours but has to be extremely precise because a wrong inference would make the program fail at runtime.

So far we have been working with single workflow configuration, which proved to be successful in typing some small but not trivial programs (*cf.* Section 5). This configuration includes all the 6 tasks described in Section 3.3, organized as described in Figure 1. In the figure each node represents a run level and the numbers in the nodes represent tasks. Each node has two outcoming links. The dotted link shows the path to be followed by the workflow in the case that the run level is successful in producing new type information. The other one is the path to follow if the run level does not find new information.



1. Generate constraints type variables for a new method.
2. Propagate minimal types.
3. Link a message send with a method.
4. Propagate maximal types.
5. Compute max concrete types from message sends.
6. Infer a minimal type from a singleton maximal type.

Figure 1. A possible workflow configuration

4. Advanced Features

One of the distinctive feature of our solution is the capability of handling blocks (also known as closures) and generic data types. Generic data types are specially useful for typing collection objects.

4.1 Blocks

A block in our type system is represented by a special type variable, which acts as a *composite* [GHJV95] type variable. Its component type variables are: the type variable associated to the return value of the block, and each of the type variables related to the parameters of the block, if the block has any.

Block type variables can be introduced either by the presence of a block literal in the AST or by a type specification. Also, when a block type variable gets involved in a subtype constraint, the other type variable is also converted to a block type variable if necessary.

When a subtype relationship is established between two block type variables, the subtyping relationship is propagated to the component variables. For example, in the following use of a block:

```
x := [ :a | a msg ] value: y.
```

In this example, some of the generated type variables are: $t_{([\ :a\ |\ a\ msg\ ])}$, $t_{([\ :a\ |\ a\ msg\ ].\uparrow)}$, t_a, t_x, t_y. And the following constraints are generated between them:

- $t_y \prec t_a$.
- $t_{([\ :a\ |\ a\ msg\ ].\uparrow)} \prec t_x$
- #msg $\in msgs(t_a)$

4.2 Generic Data Types

Our type system can handle generic data types, which are also represented by special type variables. *Generic type variables* have a subsidiary type variable. For example, if the type variable t_x is infered to be an OrderedCollection (which is a generic data type), a subsidiary variable $t_{(x.\alpha)}$ is created. The subsidiary type variable will be used to compute the type of the elements of the collection.

Generic data types are introduced by type specifications, the current algorithm does no attempt to infer that a class requires to be handled as a generic data type. Still, once introduced, generic data types can be propagated through subtype constraints. As with blocks, each time a generic type variable gets involved in a subtype constraint, the related variable is converted to a generic one, if necessary. Block type variables cannot be converted into generic data type variables or viceversa. If such situation arises, we consider it an error in the program and inform it to the user. Also, a type variable's minimal type set cannot contain both collections and other non-generic concrete types.

In our system, generic data types are *invariant*, *i.e.,* if $t_x \prec t_y$, then both subsidiary variables have to be equal: $t_{(x.\alpha)} = t_{(y.\alpha)}$. This resembles the type system of several statically-typed object-oriented languages with generic data types, such as Java [BOSW98]. Generic data types are always at class level, and not at method level.

The following example shows the constraints generated in the presence of generic data types:

```
a := OrderedCollection new.
a add:x.
y := a any.
```

Some of the generated type variables are t_a, $t_{(a.\alpha)}$, t_x and t_y. And the interesting constraints generated are:

- OrderedCollection $\in$ *min*(t_a).
- $t_x \prec t_{(a.\alpha)}$ because the type of the parameter of #add: is a subtype of the elements in the collection.
- $t_{(a.\alpha)} \prec t_y$ because the return type of #any:[4] is a subtype of the type of y.

4.3 Mixing Collections and Blocks

Since in Pharo, iterators are fully integrated in the language, collections are often combined with blocks. This combination provides some of the most challenging situations for a type inferer, but also very important to handle. Since the use of these combination is so frequent, a type inferer that is not able to handle it will be of little use for industrial programs. The following example shows the constraints that are generated in these situations:

```
| col r |
col := OrderedCollection new.
col add: anObject.
r := col sum: [ :e | e msg ].
```

In order to infer types for this piece of code, the algorithm will make use of the type specification associated to the methods #add: and #sum: of class OrderedCollection. To specify a generic data type we use a symbol instead of a concrete type, for example α, which will be associated to the subsidiary variable of the container type. So, the methods are specified as follows:

- #add: receives α[5].
- #sum: receives a block with type $\alpha\rightarrow$ SmallInteger and returns SmallInteger.

With this information, the algorithm can infer the information in Table 1.

Type Variable	Maximal Set	Minimal Set	Messages
t_{col}		OrderedCollection	#add: #sum:
$t_{anObject}$			#msg
$t_{(e\ mesg)}$		SmallInteger	
$t_{(col.\alpha)}$			#msg
t_r		SmallInteger	

Table 1. Example combining collections and blocks

5. Example

In this section we will show a small example of the execution of the inference algorithm. This example, even being an small one, it is a clear example of real industrial code. It is an implementation of a Composite Pattern [GHJV95].

[4] Which returns any element of the collection

[5] The return type is not of interest

```
Task >> initialize
    subtasks := OrderedCollection new.

Task >> addSubtask: anObject
    subtasks add: anObject

Task >> complexity: anObject
    complexity := anObject

Task >> ownCost
    ^ complexity cost: self

Task >> totalCost
    ^ self ownCost +
    (subtasks sum: [ :subtask | subtask totalCost ])
```

Figure 2. Code for the Task class.

Type Variable	Maximal Set	Minimal Set	Messages
$t_{subtasks}$		OrderedCollection	#add:
$t_{subtasks.\alpha}$	Task	Task[1]	#totalCost
$t_{(Task\gg\#totalCost.\uparrow)}$	SmallInteger[3]		
$t_{complexity}$	MediumComplexity, SmallComplexity, HighComplexity[2]		#cost:
$t_{(self\ ownCost)}$	SmallInteger		#+

Notes:

1. Task is the only implementor of #totalCost
2. The implementors of #cost:.
3. It is the result of adding two SmallInteger's.

Table 2. Results of the Task class

The figure 2 presents the code of the Task class, from a task management system. An object of this class is responsible for the calculation of its own total cost. The cost of a task depends on two things: his own cost and the sum of the total costs of his subtasks. The own cost of a task is calculated by another object that understands the message #cost: with a task as a parameter, implementing an abstract Strategy pattern [GHJV95]. There are three classes which implement #cost:, MediumComplexity, SmallComplexity and HighComplexity. The task has two instance variables, #subtasks and #complexity. Table 2 shows the most relevant information resulting from analysing the Task class.

It is remarkable that the algorithm detects the type of the elements inside *subtasks*. Once established that $t_{subtasks}$ is an OrderedCollection, we are able to take advantage of the type specification provided to OrderedCollection$\gg$#sum:. Therefore, the block has to be of type $\alpha \rightarrow SmallInteger$, where α is the type of the elements of the collection and is associ-

ated to the argument of the block (t_{subtask}). Given that t_{subtask} has to understand #totalCost and the only class implementing this method is Task, we deduce that $min(t_{\text{subtasks}.\alpha}) = \{\text{Task}\}$.

6. Discussion

One of the biggest challenges about infering types for object-oriented dynamic languages is scaling to handle big programs. Our solution has been only tested in small programs and an exhaustive analysis of its performance using more benchmarks and different kinds of programs is still a pending task. However, we are working on four different strategies to obtain acceptable execution times.

First, the possibility of specifying fixed types for core classes and other reusable libraries, reduces the number of methods to analyse and hence the number of type variables and constraints to solve.

Second, our implementation allows for different configurations, creating new tasks or even changing the existent ones. These capabilities can be exploded to regulate the relationship between precision and speed, increment the amount of information generated and even design custom tasks for a specific program.

Third, we are working on an *incremental* version of the algorithm, *i.e.,* which is able to handle method changes without running the whole type inference process from scratch. Since our intention of this work is to provide information for interactive tools, our aim is to listen to announcements regarding the method updates, discard only the type information related to the old version of the method, and keep the rest of the type information as input to the next execution of the algorithm.

Finally, one frequent problem of constraint solving type inferer is the amount of type variables that they generate, both because of their size in memory and execution time (since algorithms tend to grow exponentially on the amount of type variables). The incremental version of the algorithm will also cope with this problem, since after analysing a method we will be able to keep only the type variables in the *method interface* (*i.e.,* formal parameters and return type) discarding all other type variables which can be considered *internal* to the method.

Other approaches have proposed to use *type annotations* in order to have static type information. Type specifications show several advantages over type annotations. Since type specifications are not part of the code, the type system and the tools do not affect the structure of the code or the way a programmer works with it. In this way, our solution is oriented by the idea of *pluggable types* [Bra04], which proposes that the type information and the type system are not an integral part of the language, but only optional tools.

Moreover, most of the type specifications are not necessary for the algorithm to work. Instead, they are only a useful tool to speed up the inference process, by avoiding to analyse code that we are not intending to change. Is the programmer intends to change a portion of the core library, he could simply remove type specifications for that portion. Also, some type specifications can be the result of a previous execution of the type inferer, *i.e.,* we run the inference process on a reusable library and save the inferred types to be used as input when inferring types for a client of this library.

Our solution lacks of a formal model. The aim of this work is to provide a tool for industrial use, that can deal with real programs written in industrial languages and that is integrated in industrial programming environments. Therefore, our focus is to provide *useful* type information fast enough to be used in interactive tools. This objective can not be fulfilled if, in order to get a formal proof of soundness, we would cut off some of the most interesting parts of the language. We share this approach with renaming tool of Unterholzner [Unt12].

Our solution is able to handle generic data types but our approach can be limiting in some situations, we are analysing the idea of adding more support to Polymorphic Types: adding support to polymorphic messages, and detection of generic types developed by the programmer.

7. Related Work

To our knowledge, there are few type inference approaches combining abstract and concrete type information. In this regard, Graver [Gra89] proposed a type checking solution, based in an open world assumption. The main difference with our work, is that Graver's solution requires type specifications for class-, instance- and global variables, while in our solution this information is inferred.

Martin Unterholzner's work [Unt12] shares with ours the objective of using type inference to improve programming tools. In his case type information is used to aid method rename refactorings. To achieve his goal, he uses symbolic evaluation of the AST generated from the source code. His solution starts in a closed world context and then opens it to gather more information. His solution is focused in providing information for the renaming tool he is presenting, generating only the type information which is necessary for this specific purpose. Therefore it is not clear the applicability for use with other objectives.

Spoon and Shivers [SS04] have proposed an algorithm called DDP which prunes subgoals by giving solutions that are trivially true, reducing computation time at the cost of reducing precision. Their solution put emphasis in the performance and the scalability of the solution, but not in the precision. Our solution works also with different goals and tasks, but it does not use pruning. On the other hand our solution provides a way of inferring the type of the variable from the messages sent to it; in other words, we combine the information produced by the concrete types and the information from the abstract types.

Haupt *et al.,* [HPHf11] have proposed to gather type information by observing application code while it is running. They claim to provide more fine-grained results than type inference approaches. In this work the harvested types are objects' classes. The crucial difference with our work is that our solution is based in a static analysis, while the solution of Haupt *et al.,* gather all the information from running the tests. This approach needs to have running tests, and all the inference is based only in these tested cases.

Wang and Smith [WS01] propose an extension to CPA [Age95] which provides a way of handling generic data types. Their solution is focused in the checking of downcasts and it is highly coupled to Java Language.

Pluquet *et al.,* proposed a solution named Fast Type Reconstruction [PMW09], which is based in the static analysis of methods, but their solution only takes into account a small amount of expressions: assignation to literals, arithmetic expressions, boolean expressions and instance creations. This approach allows speeding up the solution but loses precision. Our solution propagates the return type of any message sent; providing more information about types.

Oxhoj *et al.,* [OPSb92] present a solution that can handle generic data types in the collection classes by using type variables for all the expressions and solving their relationships. However, their solution produces a high duplication of type variables, classes and methods as any generic type is duplicated for each use. For example, if there is a List of Booleans and a List of Integers; the algorithms creates two subclasses of List, one for each class of the elements.

8. Conclusion

In this work we proposed a practical solution for the lack of type information in dynamic languages, which focuses in providing useful information to be used in programming tools such as automatic refactoring, program understanding, program navigation and smart suggestions.

The solution provides an implementation for a real language in industrial level environment, which allows to build tools on top of it. The solution can be used as an external tool, without affecting the normal work of the programmer. Also, these ideas are applicable to other industrial level dynamic languages.

Regarding performance, we have promising results infering types for small programs and a testing bench for developing a responsive solution able to work on bigger programs. Our solution lacks of a formal proof of soundness, but we do not consider it a big drawback, given our objective.

One of the next steps in our research work is to add a global analysis of the type variables. For example, if the algorithm finds a local or instance variable with only one assignment, we could establish a new constraint, stronger than the subtype constraint that we are using now.

Also we intend to incorporate new tasks based on heuristics. This tasks provide potential answer with a lesser percentage of precision. Moreover, heuristic tasks could make use of fuzzy logic [ABCD65]. Heuristic tasks will be used in cases when the more precise tasks are not succesfull, adding them in a new run level. Still, other sequences of execution can be explored, changing the way the tasks are combined into different run levels.

Our solution can be used as a framework for developing different type systems. These type systems can work like plug-ins in a IDE. In this way the programmer can choose the set of tools to use in a particular problem. Also, our approach can be combined with other unrelated pluggable type systems, providing complementary information about the same program. As a result, the way different type systems and implementation cooperate and behave is an interesting approach to study.

Acknowledgements

This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council, FEDER through the 'Contrat de Projets Etat Region (CPER) 2007-2013', the Cutter ANR project, ANR-10-BLAN-0219, the MEALS Marie Curie Actions program FP7-PEOPLE-2011- IRSES MEALS (no. 295261), Universidad Nacional de Quilmes through project PUNQ 1242/13 and Universidad Nacional San Martín's through project PRI01/13.

References

[ABCD65] Peter Fisher A, Charles Arnot B, Richard Wadsworth C, and Jane Wellens D. Fuzzy sets. *Information and Control*, pages 338–353, 1965.

[ACF+13] Esteban Allende, Oscar Callau, Johan Fabry, Éric Tanter, and Marcus Denker. Gradual Typing for Smalltalk. *Science of Computer Programming*, August 2013.

[Age95] Ole Agesen. The cartesian product algorithm. In W. Olthoff, editor, *Proceedings ECOOP '95*, volume 952 of *LNCS*, pages 2–26, Aarhus, Denmark, August 1995. Springer-Verlag.

[Age96] Ole Agesen. *Concrete Type Inference: Delivering Object-Oriented Applications*. Ph.D. thesis, Stanford University, December 1996.

[Ahm13] Mian Asbat Ahmad. *New Strategies for Automated Random Testing*. PhD thesis, University of York, March 2013.

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: adding genericity to the Java programming language. In *Proceedings OOPSLA '98, ACM SIGPLAN Notices*, pages 183–200. ACM Press, 1998.

[Bra04] Gilad Bracha. Pluggable type systems, October 2004. OOPSLA Workshop on Revival of Dynamic Languages.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[Eck03] Bruce Eckel. Strong Typing vs. Strong Testing, 2003. http://www.mindview.net/WebLog/log-0025.

[Gar01] Francisco Garau. *Inferencia de tipos concretos en squeak*. Master's thesis, Universidad de Buenos Aires, Buenos Aires, 2001.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[GJ90] Justin O. Graver and Ralph E. Johnson. A type system for smalltalk. In *In Seventeenth Symposium on Principles of Programming Languages*, pages 136–150. ACM Press, 1990.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on programming language design and implementation (PLDI'05)*, pages 213–223, New York, NY, USA, 2005. ACM.

[GNDc04] Markus Gaelli, Oscar Nierstrasz, and Stéphane Ducasse. One-method commands: Linking methods and their tests. In *OOPSLA Workshop on Revival of Dynamic Languages*, October 2004.

[Gra89] Justin Graver. *Type-Checking and Type-Inference for Object-Oriented Programming Languages*. Ph.D. thesis, University of Illinois at Urbana-Champaign, August 1989.

[Hen94] Andreas V. Hense. *Polymorphic type inference for object-oriented programming languages*. Pirrot, 1994.

[HPHf11] Michael Haupt, Michael Perscheid, and Robert Hirschfeld. Type harvesting: A practical approach to obtaining typing information in dynamic programming languages. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 1282–1289, New York, NY, USA, 2011. ACM.

[Hud89] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, September 1989.

[Mar03] Robert C. Martin. Are dynamic languages going to replace static languages?, 2003. http://www.artima.com/weblogs/viewpost.jsp?thread=4639.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Opd92] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. Ph.D. thesis, University of Illinois, 1992.

[OPSb92] Nicholas Oxhoj, Jens Palsberg, and Michael Schwartzbach. Making type inference practical. In O. Lehrmann Madsen, editor, *Proceedings ECOOP '92*, volume 615 of *LNCS*, pages 329–349, Utrecht, the Netherlands, June 1992. Springer-Verlag.

[PMW09] Frédéric Pluquet, Antoine Marot, and Roel Wuyts. Fast type reconstruction for dynamically typed programming languages. In *DLS '09: Proceedings of the 5th symposium on Dynamic languages*, pages 69–78, New York, NY, USA, 2009. ACM.

[PS91] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *Proceedings OOPSLA '91, ACM SIGPLAN Notices*, volume 26, pages 146–161, November 1991.

[RL13] Romain Robbes and Michele Lanza. Improving code completion with program history. *Automated Software Engineering*, 2013. to appear.

[SS04] S. Alexander Spoon and Olin Shivers. Demand-driven type inference with subgoal pruning: Trading precision for scalability. In *Proceedings of ECOOP'04*, pages 51–74, 2004.

[ST07] Jeremy Siek and Walid Taha. Gradual typing for objects. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'07)*, volume 4609 of *LNCS*, pages 151–175. Springer Verlag, 2007.

[Suz81] Norihisa Suzuki. Inferring types in smalltalk. In *POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 187–199, New York, NY, USA, 1981. ACM Press.

[Tra09] Laurence Tratt. Dynamically typed languages. *Advances in Computers*, 77:149–184, 2009.

[Unt12] Martin Unterholzner. Refactoring support for Smalltalk using static type inference. In *Proceedings of the International Workshop on Smalltalk Technologies*, IWST '12, pages 1:1–1:18, New York, NY, USA, 2012. ACM.

[WB10] Allen Wirfs-Brock. A prototype mirrors-based refection system for javascript. https://github.com/ allenwb/jsmirrors, 2010.

[WS01] Tiejun Wang and Scott F. Smith. Precise constraint-based type inference for java. In G. Goos, J. Hartmanis, and J. van Leeuwen, editors, *Proceedings ECOOP '01*, volume 2072 of *LNCS*, pages 99–118, Budapest, Hungary, June 2001. Springer-Verlag.

Benzo: Reflective Glue for Low-level Programming

Camillo Bruni Stéphane Ducasse
Igor Stasenko
RMoD, INRIA Lille - Nord Europe, France
http://rmod.lille.inria.fr

Guido Chari
Departamento de Computación, FCEyN, UBA
and CONICET
http://lafhis.dc.uba.ar

Abstract

The goal of high-level low-level programming is to bring the abstraction capabilities of high-level languages to the system programming domain, such as virtual machines (VMs) and language runtimes. However, existing solutions are bound to compilation time and expose limited possibilities to be changed at *runtime and from language-side*. They do not fit well with fully reflective languages and environments.

We propose Benzo[1], a lightweight framework for high-level low-level programming that allows developers to generate and execute at runtime low-level code (assembly). It promotes the implementation, and dynamic modification, of system components with high-level language tools outperforming existing dynamic solutions.

Since Benzo is a general framework we choose three applications that cover an important range of the spectrum of system programming for validating the infrastructure: a Foreign Function Interface (FFI), primitives instrumentation and a just-in-time bytecode compiler (JIT). With Benzo we show that these typical VM-level components are feasible as reflective language-side implementations. Due to its unique combination of high-level reflection and low-level programming, Benzo shows better performance for these three applications than the comparable high-level implementations.

Categories and Subject Descriptors D.3.3 [*Programming Language*]: Language Constructs and Features; D.3.2 [*Programming Language*]: Language Classifications—Very high-level languages

Keywords system programming, reflection, managed runtime extensions, dynamic native code generation

1. Introduction

High-level low-level programming [16] encourages to use high-level languages such as Java to build low-level execution infrastructures or to do system programming. Frampton et al. present a framework that is biased towards a statically typed high-level language, taking strict security aspects into account. It is successfully used in experimental high-level self-hosted virtual machines (VMs) such as Jikes [3].

The results presented by Frampton et al. are inspiring for high-level system programming developers. Their approach promotes to tackle low-level system programming tasks with the tools and abstractions of high-level languages. However, the solution has certain limitations when applied to a dynamic and reflective context.

By the term "dynamic and reflective" we refer to combined reflective capabilities for a language to inspect (introspection) and change its own execution (intercession) at runtime [24].

The most important limitation we found when approaching dynamic high-level low-level programming attaining to the existent solutions is the following:

It is not possible to generate ad-hoc native code (assembly) at runtime and execute it dynamically.

We illustrate it with the following use case.

1.1 Use Case: Dynamic Primitive Instrumentation

To illustrate this limitation we take the example of dynamically intercepting VM primitives. In most managed runtimes, VM primitives are used for essential tasks such as object creation or provide fundamental functionality that can not be obtained otherwise at language-side [17, page 52].

Already the simple example of measuring the time spent in an essential primitive while executing a performance critical task is difficult to do efficiently.

Reflective solutions at language-side could take advantage of the intercession capabilities that allow changing or augmenting almost any behavior at runtime. In practice, this does not work for primitives. In addition, reflectively measuring the duration of the time primitive itself will easily cause meta-recursion. Hence in general reflective instrumentation will not work on primitives used for the instrumentation itself. Thus efficiently instrumenting VM primitives in a dynamic and reflective environment is not feasible in many cases.

Naturally if we leave the high-level realm there are efficient tools at hand for instrumentation. Existing solutions for efficient low-level instrumentation such as DTrace [12] work by installing static hooks. Though in a reflective language assumptions can change at runtime and thus static solutions are not appropriate. At the same time we clearly see that such instrumentation is not a typical high-level application. To combine these two worlds we need a different approach.

[1] The name Benzo originates from Benzocyclobuten which is an organic glue used in wafer production.

Approaching Dynamic Primitive Instrumentation with Benzo. Using Benzo framework for *reflective low-level programming* we show in Section 4.2, as one of three proof of concepts, a solution to the problem of efficient dynamic VM primitive instrumentation. By dynamically generating and activating native code from language-side we are able to create customized primitives. With Benzo even essential primitives can be dynamically change without the need of a system restart.

Although this is a clear example, Benzo is a general reflective high-level low-level programming framework that overcomes also other system programming limitations. We illustrate its advantages with other distinct examples such as a Foreign Function Interface (FFI) and a just-in-time bytecode compiler (JIT) in Section 4.

1.2 Bridging Abstraction Layers

Extending high-level language runtimes is difficult due to their static low-level construction which usually shares little resemblance with the language-side. Yet for tasks, like the previously presented primitive instrumentation, if we want to tackle it with a high-level language, we need solid low-level interaction.

Requirements. In Section 2 we describe the solutions offered by traditional approaches to modify or extend a language runtime: language-side libraries, reflective capabilities, VM extensions or hybrid approaches. However, none of them is powerful or general enough to support our use-case. A general and uniform solution is needed that spans over several abstraction layers. It has to interact on a high-level with the reflective capabilities of the language runtime and at the same time provide an interface to interact with low-level code. To stay flexible and compatible enough the solution should add these new key features with as little static low-level intrusion as possible.

- It must be *reflective* in the sense it must support *dynamic* changes of the language runtime (VM) without requiring a system restart.
- It should imply minimal changes to the existing low-level runtimes to *considerably reduce development efforts*.

Benzo a Framework for Reflective Low-level Programming. High-level low-level programming is a powerful technique for system programming without resorting to static low-level environments [16, 34] that almost fulfills our requirements. However in a reflective setup it fails to comply with the first requirement mentioned in the previous paragraph.

Our approach consists of Benzo, a lightweight, dynamic and reflective framework that tackles the exposed limitations. Benzo dynamically generates native code from language-side and can execute the changes in place. It relies only on a small set of generic VM extensions described in Section 3.1.

Framework applications. In Section 4 we advocate the contribution of this approach by providing three different incremental examples that heavily use the framework from language-side. They rely on it for extending or even improving language runtime capabilities. They consist of:

FFI A complete language-side Foreign Function Interface (FFI) implementation, described in Section 4.1.

Dynamic Primitives A language-side compilation toolchain that replaces system primitives at runtime with customized code, described in Section 4.2.

Language-side JIT Compiler A JIT compiler that works at language-side and interacts with the VM for code synchronization, described in Section 4.3

As illustrated by these three distinct examples, the contributions of this paper are:

- Encouraging the extension of high-level language runtimes through the use of reflective low-level programming promoting an open interaction with the low-level world without the overheads imposed by high-level one.
- A proof of concept of the proposal with the implementation and description of three different tools that heavily use reflective low-level programming and covers distinct scenarios.

2. Current Approaches for Modifying/Extending Runtimes

We present now an overview of the approaches used to extend a language runtime and expose their limits.

High-level languages are in general sustained by a VM and a vast set of libraries written in the language itself. Extending or improving the existing Runtimes is a difficult task. In most cases the VM is considered as a black box. Additionally the VM is written in a completely different language using another abstraction level than the one it supports. Typically high-level language VMs are written in C or C++. To address runtime extensions in this context there exist some known approaches:

Language-side Library based on implementing a new or existing library.
Language-side Reflective Extension relying on reflective features of the language.
VM Extension by writing plugins or changing the core of the VM.
Hybrid Extension by accessing external libraries using FFI.

The relation between the side concerning the abstraction and implementation levels (VM vs. Language) of these extensions is illustrated in Figure 1.

2.1 Language-side Library

The most straight forward solution for extending a language is to write libraries within the language itself. This option provides the advantage that the aggregate behavior is accessible and evolvable for any language developer.

However language-side libraries are constrained by the underlying managed runtime. The VM separates the language from the low-level internal details. As a consequence language-side libraries are not feasible for all feature requirements. For instance the previously mentioned example of instrumenting the runtime is not possible as a standard language-side extension without a considerable performance loss. So, even though we prefer extensions and optimizations at language-side, there are certain limitations of a managed runtime that can not be circumvented. If all language-side optimization opportunities have been exhausted it is exposing the need to resort to lower level approaches.

Language-side libraries are constrained to the capabilities of the underlying VM and thus not general enough. Addi-

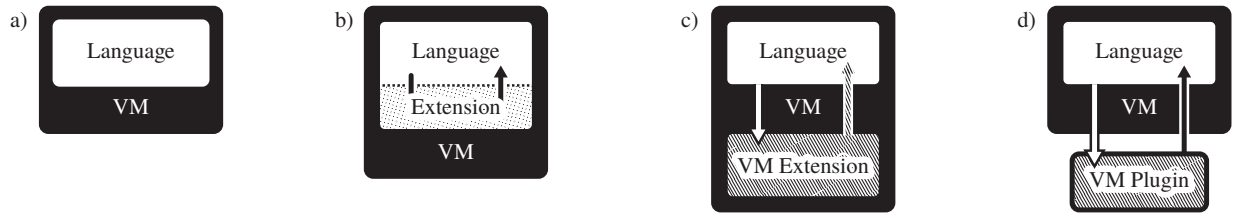


Figure 1: Comparing different extension mechanisms: a) language running on a standard VM, b) language-side implementation of an extension c) language using features from a VM extension, d) language using features from a VM plugin.

tionally not all performance bottlenecks can be addressed at language-side.

2.2 Language-side Reflective Extensions.

This is a subcase of the previous approach but in the context of reflective environments that expose particular characteristics.

For instance, Meta Object Protocols (MOP) [22] based on reflection [24] are used to define certain control points in the system to change the language. By composing meta objects it is possible to even modify the semantics of the language. Several languages such as Smalltalk, Python, and others provide reflective capabilities with different depths [4, 15, 32].

However most modern programming languages only have very limited support for intercession. Hence the possibilities for dynamically changing language semantics or features are limited. Furthermore reflective capabilities are hard to implement efficiently. Reflection imposes substantial performance penalties on most computations by postponing bindings [25].

Nevertheless there are exceptions for a subset of reflective behavior which are implemented efficiently using a high-level MOP [33]. Though these approaches remain as a few exceptions. In the typical low-level VM it is difficult to gain reflective access to language-side objects.

Similar to the previous case, our goal is to extend language features in a general way and it was shown that this is only partially possible by reflective extensions.

Reflective capabilities are not enough for general extensions. Even when suitable, they usually pose a significant performance overhead up to the point where they become unfeasible.

2.3 VM Extensions

Another approach is to attach plugins to the VM. Plugins are direct bindings to external libraries described at VM-side or libraries linked to the VM executable [8, Ch. 5]. They provide a performance boost in comparison to pure language-side solutions. Using highly optimized native libraries it is straightforward to outperform code written at language-side.

However, plugins are commonly written in the same language as the VM, at a low abstraction level. Few exceptions are self-hosted languages [29, 31, 34]. To support a fluent development process, VMs should come with an infrastructure for building extensions at same abstraction level than the language. Instead they tend to be very complex and to have sluggish building processes. For example, only a few VMs have high-level debugging facilities [19, 31, 34]. Also from a VM maintenance point of view, extensions have to be avoided if possible and should only be used for critical performance issues that can not be properly addressed at language-side. An example of how the complexity of the VM can affect development efforts is the core of the Self VM [30]. After reduced development resources parts of the complex but efficient compiler infrastructure had to be abandoned in favor of a more maintainable code-base.

VM extensions provide good performance but imply resorting to low-level tools where abstraction advantages of high-level languages are restricted.

2.4 Foreign Libraries

The last approach is to reuse an existing library usually implemented in a foreign language. The languages interact through a well-defined Foreign Function Interface (FFI). FFI-based extensions are an hybrid approach between pure language-side extensions and VM-side ones. Interaction with native libraries is supported by a dedicated VM functionality for calling external functions. This allows for a smooth interaction of external code and language-side code. FFI based extensions share the benefits of a maintainable and efficient language-side library with modest implementation efforts.

However, FFI is only a bridge or interface for allowing the interaction of different languages. It is not possible to directly synthesize new native features from language-side. For this purpose we have to interact with a custom-made native library. From an extension point of view this is close to the VM extensions discussed previously.

Additionally to the interface limitations, there exists a performance overhead in FFI for making the interaction between different languages possible. This is due to marshalling arguments and types between both languages [14, 28].

FFI allows developers to cross language-barriers with less effort than a VM extension and enables a tight integration with existing libraries. However, it is only an interface and depends on extensions already available. Moreover, performance penalties are considerable in some cases.

2.5 Summary

The approaches discussed above rely on language-side code with the exception of VM extensions. However we have shown limitations for all of them. Hybrid solutions such as FFIs are the only ones that come close to meet all our requirements. The only downside of FFIs is that it is not possible to directly synthesize new custom native functionality with them. Hence, for our purpose FFIs are not general enough as it is not possible to solve our initial use case for dynamically instrument primitives.

Benzo takes the advantages from the presented approaches but avoids their weaknesses. By using Benzo high-level developers tackle the problems in a uniform way by exploiting the debugging and development facilities provided by the language. Developers model their applications or libraries with a high-level language and have a clear interface for generating and executing efficient low-level code when needed, but inside the same language and with the same abstractions and reflective capabilities. In Section 5.2 we show that we achieve the performance requirements of low-level environments.

3. Benzo Implementation in a Nutshell

This section covers the necessary changes to make Benzo compatible with the Pharo VM and the language-side behavior contributions.

3.1 VM Context

Pharo is a Smalltalk dialect that emerged from the Squeak project [19]. The Pharo VM implementation [27] also evolved from the original Squeak bytecode interpreter. The current VM uses a moving Garbage Collector (GC) with two generations. Additionally it efficiently maps Smalltalk method activation context to stack frames. The VM uses a JIT that maps bytecodes to native instructions and applies basic register allocation to reduce stack load. This situation is not a direct requirement for Benzo but it is assumed as given and thus not further discussed in detail.

However Benzo requires certain features that were not supported in the existing VM implementation. Mainly we need to generate executable code at runtime and run it. This requirement is essential and applies to any VM that wants to support dynamic code execution at runtime.

3.1.1 Executable Memory

We use standard Smalltalk objects to hold the generated native code. However, by default the object memory is not executable. This leaves two choices: mark the whole object memory executable or only move the objects with the native code to a special executable memory region. We took the path of least resistance and marked the whole object memory as executable. The other solution requires substantial changes for memory management. As the VM has a moving GC we only access high-level Smalltalk objects via an indirection from low-level code.

Another approach would have been to harness the fixed sized executable region used by the existing JIT. However the JIT space does not hold normal Smalltalk objects but special low-level structures and uses its own special GC.

3.1.2 VM Interaction

The standard way in Smalltalk to execute low-level code is to use a tag in the method definition. The following example shows such a method on the `Float` class.

```
* aNumber
    <primitive: 49>
    ↑ aNumber adaptToFloat: self andSend: #*
```

Here we use the primitive 49 to call a VM function which efficiently multiplies two floats. Figure 2-a describes the case where the primitive is successfully executed. However if the primitive is unable to do the operation, for instance if the argument `aNumber` is not a float, it will signal a failure which causes the VM to execute the fallback Smalltalk code in the method body. Fig. 2-b describes it.

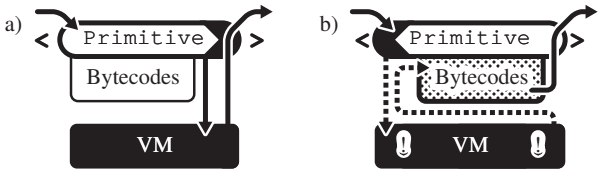


Figure 2: Generic primitive methods in Pharo: a) A primitive completely bypasses the bytecode, b) A failing primitive executes the Smalltalk bytecode as fallback.

Benzo uses the primitives as a gate to enter the low-level world from the language-side. The primitive then executes the native code generated and returns to language-side. The generated native code is appended inside the compiled method object. When the primitive is activated, it accesses the currently executed compiled method via a VM function. Figure 3 shows the structure of a Smalltalk compiled method that has native code attached to it. We see the primitive tag on top, followed by the literal frame which holds references to symbols and classes used in the method. The subsequent Smalltalk bytecode is the fallback code executed only if the primitive fails. Only then appears the native instructions. A marker at the end of the compiled method called trailer type is used to flag methods that actually have native code attached to them.

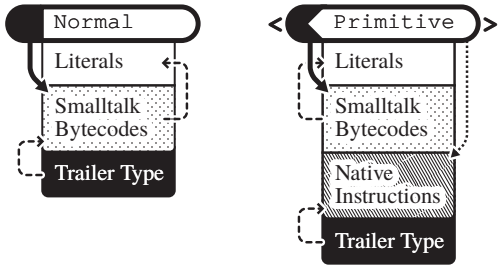


Figure 3: A standard Smalltalk compiled method on the left and a method with appended native instructions generated by Benzo.

Since compiled methods are first-class objects it is possible to modify them at runtime and append the native code. The primitive `primitiveNativeCall`, which is implemented by Benzo, is the responsible of running the native instructions in a Smalltalk method. The code example `interrupt3` shows a very basic application of our infrastructure. In Section 3.2 we describe how Benzo uses Smalltalk code to generate the native instructions, specifically Section 3.2.1 will explain more detailed examples.

```
interrupt3
    <primitive: 'primitiveNativeCall'
     module: 'BenzoPlugin' >
    Benzo generate: [ :asm | asm int3 ]
```

Listing 1: Smalltalk method using Benzo for low-level debugging.

3.1.3 Native Code Platform Interaction

To ensure that the code is compatible with the current platform a VM specific marker is expected at the beginning

of the native code on the compiled method. Upon activation Benzo compares this marker with the one from the current VM. If they don't match, Benzo signals a failure that causes the VM to evaluate the fallback Smalltalk code. With this elegant approach Benzo regenerates native code lazily on new platforms. Moreover, it does not have to flush the native code when the application is restarted on the same platform.

3.1.4 Garbage Collector Interaction

Compiled methods in Pharo have a special section, the literal frame, which stores objects referenced in the bytecodes. Bytecodes then only have indirect access to these objects by indexing into the literal frame. This simplifies the implementation of the garbage collector as it only has to scan the beginning of each method for possible references to objects. So the GC only tracks Smalltalk objects when they are in the method's literal frame.

The moving GC of the VM used for Pharo has a significant impact on the low-level code we can generate using Benzo. For instance it is not possible to statically refer to language-side objects from native code as object addresses changes after each garbage collection. Modifying the GC to support regions of non-moving objects would solve this problem. However we chose to minimize the number of low-level VM modification necessary to run our experiments and opted for a simpler solution. Like the existing compiled methods, Benzo's accesses language-side objects through an indirection.

For indirectly accessing objects the Pharo VM already features a special structure, named external roots. This array has a fixed-location in memory which can be used to access moving language-side objects. The GC updates the addresses in this VM structure after each run. Hence we have the static address of the external roots object as an entry point to statically access a Smalltalk objects.

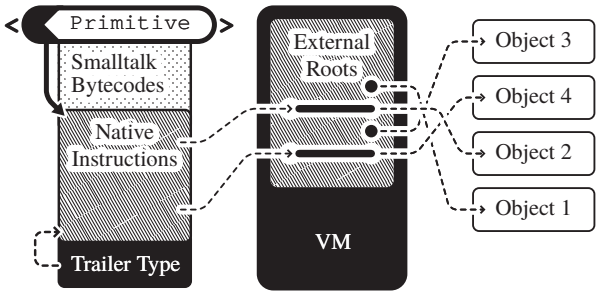


Figure 4: Pointers to objects registered as external roots are pinpointed at fixed offset in global VM-level object.

So for accessing Smalltalk objects within native code we first register it as an external root object and access it only indirectly. This means that for native code, instead of a method-local literal array we share a global literal array as shown in Figure 4. Benzo only adds an `Array` to the external root objects which is managed from language-side and administers all references.

3.1.5 JIT Interaction

When the Pharo VM starts the execution of dynamic generated code the execution environment changes slightly. Similarly, when entering primitives or plugin code that mode is left and a normal C level execution environment is reestablished until the primitive finishes and the VM jumps back to the jitted code. To avoid this context changes that imply a considerable performance overhead, we extend the VM to support inlining of native code in the JIT phase following the same strategy as other existing primitives which are inlined at JIT-level.

The Benzo prologue and epilogue used for managing the low-level stack are replaced by an adapted version for the JIT. The performance boost of this optimization is further discussed in Section 5.2.

3.1.6 Error Handling

Benzo provides an error handling facility that allows to return high-level error messages from the low-level code. The native code builder provides a helper method called `failWithMessage:` that generates the proper assembler instructions to return a full error message. This allows plugins to return clear and meaningful error codes, improving the debugging tasks and enabling a better interaction with users.

3.2 Benzo's Language-Side Implementation

We keep the interface to the low-level world minimal. The following describes the salient features in the high-level language-side of Benzo.

3.2.1 Code Generation

Benzo delegates native code generation to a full assembler written in Smalltalk. The following example shows how to use the assembler to generate the native code for moving `1` into the 32-bit register `EAX`.

```
ASM x86 generate: [ :asm |
    asm mov: 1 asUImm to: asm EAX ].
```

The implementation first creates a slightly more abstract intermediate format. The abstract operations can be extended by custom operations that may expand to several native instructions. For pragmatic reasons current implementation only supports `x86` and `x86-64`.

The plan is to improve the platform independence by implementing a more abstract domain specific language for Benzo low-level instructions.

The full runtime features of Pharo are available when generating native code. Hence complex instruction sequences can easily be delegated to other objects. In the following example we use a VM helper to instantiate an array, note that these are all standard Pharo message sends:

```
ASM x86 generate: [ :asm :helper | | register |
    register ← helper classArray.
    register ← helper
        instantiateClass: register
        indexableSize: 10
    asm mov: register to: asm resultRegister.
    ].
```

In this case the `#instantiateClass:indexableSize:` will generate the proper native code to call to a VM function and make sure that the side-effects of a possible GC run are handled properly. By default the value in the result register is returned back to the image, on `x86` this defaults to `EAX`. The VM helper exposes a basic, low-level interface to access objects and its properties. Additional methods cover the access of external roots described in Section 3.1.4. Section 4 will give more complete applications which are based on Benzo.

3.2.2 Code Activation

Benzo primitive is responsible for the native code activation which consists of three main steps:

- Check if there is native code in the actual compiled method and if it is compatible with the current platform.
- Generate native code if necessary.
- Activate the native code for execution.

The example in Listing 1 uses Benzo's generator to create and install the native code which would trigger a low-level interrupt. Behind the scenes Benzo adds some more information to the code as the already mentioned platform marker. For activation Benzo uses reflective features to restart the method containing the native code. Upon the second activation, after already generating the native code, Benzo moves the native code to the end of the compiled method and activates it. This mechanism is shown in Figure 5.

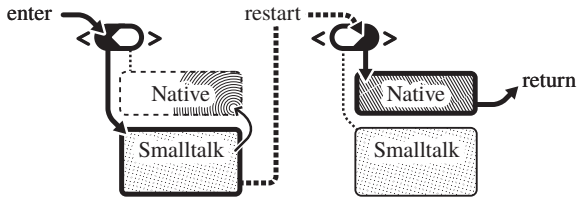


Figure 5: Native code activation with Benzo: The first call triggers the code generation. Then the method is restarted and the native code executed.

4. Benzo in Practice

In the following Section we will present a dynamic language-side implementation, based on Benzo, for each of the three examples mentioned in the introduction for extending language runtimes.

4.1 NativeBoost: a Benzo-based Foreign Function Interface

FFIs enable a programmer to call external functions without the need to implement additional VM extensions. NativeBoost [11] is a full FFI developed on top of Benzo. An FFI implementation consists of two main parts:

- **Execution:** calling external functions.
- **Marshalling:** converting data between the languages.

Typically most of these two parts are implemented in the VM with statically defined bindings to convert basic types such as integers, strings and floats between the different representations. Furthermore they provide entry points to find external functions by name in a certain external library. Relying on Benzo capability to dynamically generate and execute native code we developed a complete FFI at language-side. This way the VM no longer requires to have a specific FFI extension.

FFI at Language-side. The fact that via FFI we can call external functions makes it a perfect option to replace VM extensions defined at low-level side since FFI relies only on one generic low-level extension: the language-side has to be able to generate and subsequently call native instructions. A VM with a well-defined plugin infrastructure enforces the same level of separation. However unlike plugins, FFI bindings are implemented without crossing a language barrier. Most code for FFI bindings can be written at language-side in already existing familiar infrastructure. Furthermore, compared to a low-level plugin, a language-side library is easier to evolve and maintain.

FFI-based language extensions also provide a certain level of portability. Often the only artifact that has to be ported is the FFI plugin for the VM. In the optimal case the high-level FFI code is completely compatible. If the platform does not provide the same signature for the function, only the language-side code requires changes. This is preferable since the language-side part of the FFI code relies on better abstractions and infrastructure for debugging.

NativeBoost does not even depend on a specific VM plugin but on the generic infrastructure provided by Benzo. All the FFI is implemented at high-level language-side. Figure 6 shows how only the last step in calling an external function relies on low-level VM interaction. Section 4.1.2 explains the execution component details. Via reflection techniques NativeBoost provides a simple yet powerful marshalling library which is further described in Section 4.1.3.

4.1.1 NativeBoost in a Nutshell

A very simple example to illustrate the functionality of NativeBoost is to access the current environment variables. We do this by calling the `getenv` and `setenv` C functions. `getenv` takes a name as single argument and returns the value of that environment variable as a string.

```
getenv: name
    ↑ FFI call: 'String getenv(String name)'
```

In this example NativeBoost automatically detects that the arguments for the Smalltalk method are the same as for the low-level C function. The most important aspect about this example is that it is written with standard Smalltalk code. In figure 5 we show how NativeBoost lazily generates native code on the first method activation.

4.1.2 External Functions and Symbols

For a complete and practical FFI implementation the gathering of external function addresses is an imperative requirement. NativeBoost supports this for every platform. For instance, on UNIX-like systems NativeBoost achieves this by wrapping around the existing functions `dlopen`, used for opening shared libraries, and `dlsym`, used for returning function name addresses.

4.1.3 NativeBoost Symbiosis with Pharo

NativeBoost uses reflection capabilities to detect and marshall Smalltalk method arguments to C-level function arguments taking advantage of the full power of Smalltalk to support complex type conversions. This allows to have simpler declaration of FFI calls.

Argument Detection. NativeBoost automatically detects the arguments for the C function from the name given in its declaration. For instance, in the example of Section 4.1.1 the argument for `getenv` is found by looking at the method source code. In more complex setups the arguments of the method might not correspond to the order of the C function's arguments and a binding by name does the job.

Type Marshalling. NativeBoost automatically converts primitive types between the C world and Smalltalk. In the same previous example of `getenv` we replaced the `char *`

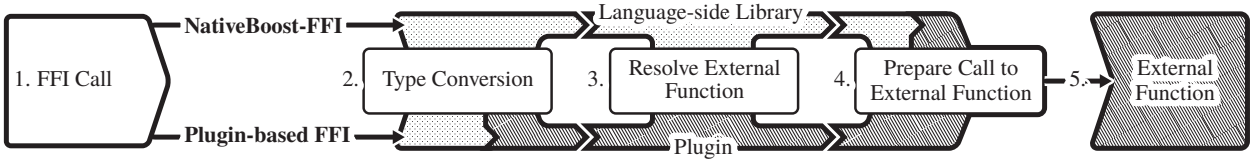


Figure 6: NativeBoost Overview: Unlike typical FFI implementations NativeBoost only resorts to the VM-level when actually calling the external function in step 4. Typical implementations already cross the low-level barrier during the type conversions at step 2.

from the original function signature with the single type `String`. This allows NativeBoost to automatically marshal the Smalltalk String into the corresponding C representation. From a Smalltalk point of view the original declaration `char *` is ambiguous. Smalltalk distinguishes between arrays of characters and a real string. For more elaborate type conversions such as C-level structs NativeBoost uses marshalling objects that reify the low-level common structures.

4.1.4 NativeBoost Performance

Compared to a static plugin-based FFI implementation NativeBoost has only a one-time startup overhead with its numbers shown in Section 5.2. Generating the native code at language-side is substantially slower than directly setting up all the conversions and calling the external functions from C code. In some cases the penalty for some compilation effort on NativeBoost is as high as a factor of 100 compared to classic approaches. Under the assumption that the method is called several times this overhead may be considered negligible. The following table shows a performance comparison of three different FFI implementations for Pharo Smalltalk.

	Call Time	Relative Time
NativeBoost	`10.53` ± `0.35` ms	1.0×
Alien	`31.09` ± `0.94` ms	≈ 3.0×
FFI	`19.55` ± `0.64` ms	≈ 1.9×

Table 1: Different FFI implementations in Pharo running `abs` with a single argument. Alien does marshalling at language-side while FFI does everything in C.

Table 1 measures the accumulative time of 100'000 FFI calls. Included in these numbers is at least one additional Smalltalk message send to activate the NativeBoost method containing the actual call to the C function. Each benchmark itself is run 1000 times and the average and standard deviation is taken. We also measured calls with more complex type conversions where the performance boost against Alien pronounced even more because NativeBoost's language-side marshalling is nativized. The JIT interaction described in Section 3.1.5 is also an important optimization factor especially when calling out small helper routines where the context switch from jitted mode is not negligible.

4.2 Reflective Primitives

Pharo VM is developed in a language that is a subset of Smalltalk known as Slang, which is transformed to C and then compiled using a standard C compiler. Slang basically has the same syntax as Smalltalk but is semantically constrained to expressions that can be resolved statically at compilation or code generation time and are compatible with C. Hence Slang's semantics are closer to C than to Smalltalk. The primitives of the language are written in Slang since are part of the VM.

That's why even in highly reflective languages like Smalltalk where almost every aspect of the language is available for inspection or modification [13] primitives can not be changed at runtime. Waterfall [2] is a JIT compiler that takes the standard primitive definitions from the code written in Slang and translates them to native code. This replaces the indirection via C that is used in the default compilation process for primitives. But given that Slang source code can be modified at runtime as any other Smalltalk method, Waterfall fosters primitives to be dynamically changed.

4.2.1 Waterfall Compiler Summary

From a high-level point of view the services provided by Waterfall can be outlined in two main functionalities:

- Compile Slang code on demand (lazily), at runtime and from language-side.
- Provide a clear interface for executing, also at runtime and from language-side, the native code generated by the compiler.

The first item allows to change the code of primitives at language-side and generate the corresponding native code when needed. Also it provides the potential to write methods or functionalities with the same Smalltalk syntax but with a static semantic. It consists essentially of a transformation toolchain that uses the AST that is generated by the standard Pharo compiler harnessing that Slang and Smalltalk have the same syntax. Then this AST representation is translated to native code enforcing C-like Slang semantics. The current prototype has only three fully implemented stages: Slang to AST, AST to an IR (between TAC and SSA) and finally AST or IR to native. The design is open for future additions at any level. One typical enhancement missing is having different levels of intermediate representations with various techniques on code optimization and register allocation strategies as modern compilers propose [5, Ch. 1].

The second item from above list is the responsible of presenting a clear interface that allows executing the dynamically generated native code. This includes for instance the gathering of positions of the VM internal symbols. Waterfall relies on NativeBoost, the Benzo-based FFI presented in the previous section, for interfacing with C libraries (`dlsym`). It also includes the linking between the two worlds: Smalltalk and native. Benzo is heavily used for providing this second main functionality.

Primitives in Smalltalk. As already partially explained, whenever a method is compiled with the `primitive` pragma

as shown in Section 3.1.2 a flag is set on the `CompiledMethod`. If the VM finds that the flag is set, it gets the number of the primitive and instead of interpreting the bytecodes it calls the corresponding function at VM-level[17]. The binding between primitives and numbers is described in a table indexed by number.

Smalltalk distinguishes two types of primitives: essential and non-essential primitives. Essential primitives are required for the bootstrapping and the essential operations of the language, such as creating a new object or activating a block. The second category of primitives are mainly used for optimization purposes.

Dynamically Interchangeable Primitives. Waterfall uses Benzo's mechanism for replacing primitive methods with customized nativized versions that are created dynamically as described in Section 3. This loophole of the language exploited by Waterfall provides the advantage of having the possibility to dynamically modifiy some VM behavior with a considerable much lower penalty on performance.

4.2.2 Benefits and Contribution

We identified two main benefits of changing VM primitives at runtime:

- Reducing VM complexity by implementing non-essential primitives reflectively at language-side.
- Dynamic instrumentation of primitives.

Reducing VM Complexity. In Section 2 we concluded that VM extensions are only justified in the presence of strong performance requirements. All non-essential primitives fall into that category. Using Waterfall these primitives may be implemented at language-side. This means that these primitives become first-class citizens of the high-level environment and thus evolve with less effort.

Instrumentation of Primitives. Essential primitives can not be fully implemented at language-side using Waterfall. These primitives are required for system startup. Hence they would trigger an endless recursion when booting up the system. However nothing prevents from replacing essential primitives at runtime with customized versions. We use Waterfall with primitives for efficient instrumentation purposes.

Actually it is absolutely possible to do instrumentation completely at language-side for non-essential primitives without Waterfall by accepting the performance penalty, but for essential primitives doing it is a very fragile task. The chances of accidentally invoking the same primitive in the language-side instrumentation code are high. Without very careful design the instrumentation code will thus trigger an endless recursion. Also performance issues could be prohibitive for language-side solutions. With Waterfall we can avoid these issues since the instrumentation code eventually will be implemented at the lowest level.

4.2.3 Performance Analysis

For comparing performance we implement a very simple integer operation primitive (>) using three different approaches. The first approach is the implementation with Waterfall. The second is to run the language-side implementation that is triggered whenever the standard primitive failed. Finally the fast standard primitive provided by the VM. We run the three approaches by measuring the cumulative time over one million primitive activations averaged over 100 runs. The absolute numbers are less important than the relative factor between them. We present the results of this experiment in Table 2.

	Running Time	Relative Time
VM	`6.4` ± `0.14` ms	1.0×
Waterfall	`22.8` ± `0.17` ms	≈ 3.6×
Reflective	`195.0` ± `0.16` ms	≈ 30.0×

Table 2: Comparing running time of different implementations of integer arithmetic primitive.

As expected Waterfall's solution outperforms pure reflective one by factor 9 to 10. Waterfall clearly outperforms a purely reflective solution since all the meta programming overhead for the intercession mechanism is avoided. This results thus makes a whole new set of runtime extensions feasible that were previously limited by their strong performance penalty. Furthermore the performance penalty over a completely optimized VM solution that has extreme optimization techniques, such as inlining and register allocation, is less than a factor of 4. Applying standard optimization techniques, not yet implemented in Waterfall, will almost sure improve these numbers even more.

4.3 Nabujito JIT Compiler

In this section we present Nabujito, a Benzo-based approach for a language-side JIT compiler. Nabujito goes even further than Waterfall using almost the same techniques. However instead of focusing on primitives, Nabujito generates native executable code for standard Smalltalk methods. Primitives tend to be more low-level, whereas Nabujito focuses on high-level Smalltalk code.

4.3.1 The JIT of the Pharo VM

The Pharo VM already comes with a JIT that translates bytecodes to native instructions. It transforms Smalltalk methods into slightly optimized native code at runtime. The main speed improvement comes from avoiding bytecode dispatching and by inlining certain known operations and primitives [6].

The most complex logic of the JIT infrastructure deals with the dynamic nature of the Smalltalk environment. `Methods` and `classes` can be changed at runtime and that has to be addressed by the JIT infrastructure. The JIT compiler, by which we refer in this context to the transformation of bytecodes to native code, represents a small part of the whole infrastructure. There exists more important stages as an additional register allocation pass to reduce the number of stack operations [26, 27]. The existing JIT infrastructure is implemented in Slang [8, Ch. 5] as the rest of the VM.

4.3.2 Limitations of Standard JIT Compilers

Since the JIT compiler itself is quite decoupled from the rest of the JIT infrastructure we believe that a hard-coded static and low-level implementation is not optimal for several reasons:

- Optimizing Smalltalk code requires strong interactions with the dynamic environment.
- Accessing language-side properties from the VM-side is hard.
- Changing the JIT compiler requires changes to the VM code.

- The JIT reimplements primitives for optimization reasons resulting in code duplication.

Optimizations Limits for Smalltalk. In Smalltalk methods tend to be very small and it is considered good practice to delegate behavior to other objects. That implies that several common optimization techniques for static languages do not work. The dynamic method activation do not provide enough context for a static compiler to optimize methods. Hence after inline caches and register allocation the next optimization technique is inlining. However inlining in a dynamic context is difficult and requires hooks at VM level to invalidate native code when the language-side changes. Since in Smalltalk compiling a method is handled completely with language-side code most of the infrastructure to get notified about method changes is already present.

Primitives in the Existing JIT. The existing JIT reimplements the most used primitives at VM-level. This is necessary for instance to guarantee fast integer operations. A typical example is the integer addition which has to deal with overflow checks and conversion of tagged integers. In Section 4.2 we describe how Waterfall suffers a similar requirement. Hence Waterfall manually defines such primitives in terms of native assembler instructions through the language-side Benzo interface. Nabujito, a language-side JIT compiler described on next section, reuses the same optimized primitives so we rely on a single optimized definition which is shared amongst all native code libraries.

4.3.3 Implementing Nabujito

Nabujito is an experimental JIT implementation which replaces the bytecode to native code translation of the existing JIT infrastructure with a dynamic language-side implementation. Nabujito is implemented mainly with a visitor strategy over the intermediate bytecode representation. Additionally we reimplemented using Benzo vital native routines for the JIT which are not directly exported by the VM.

Nabujito relies on the following VM-level infrastructure to manage and run native code:

- Fixed native code memory segments.
- Routines for switching contexts.
- Native stack management.

Dynamic Code Generation. To simplify the implementation we decide to manually trigger JIT compilation. For primitives known by Waterfall we rely on that infrastructure to generate the native code. For standard methods Nabujito takes the bytecodes and transforms them to native code.

It also applies optimizations such as creating low-level branches for Smalltalk level branching operations like `ifTrue:`. Optimizations for additional methods are all implemented flexibly at language-side. Wherever possible we reimplement the same behavior as the existing native JIT compiler.

Eventually the native code is ready and Benzo attaches it to the existing compiled method. When the language-side jitted code is activated Benzo ensures that we do not have to leave the JIT execution mode, and thus we can call methods at the same speed as the existing JIT. The benchmarks of section 4.3.4 show the empirical results.

4.3.4 Nabujito Performance

Performance is of course the main contribution of a JIT and it is imperative to analyze the efficiency of a language-side implementation.

Nabujito essentially generates the same native code as the VM counterpart. For the experiment we reimplement the C routines found in the VM JIT at language-side. There is no speed difference in the generated native code. However Nabujito is slower during the warm-up phase. Compilation of the native instructions will take considerably more time compared to the C implementation of the same bytecode to assembler transformation. However this is not critical for long-term applications.

	Compilation Time
Pharo Compiler	`71` $\pm$ `1` ms
Nabujito	`73` $\pm$ `1` ms

Table 3: Compilation efforts of the standard Smalltalk compiler in Pharo and Nabujito for the a simple method returning the constant `nil`.

In Table 3 we compare the compilation speed of the standard Pharo compiler and Nabujito. We measure the accumulated time spent to compile the method 1000 times. The average and deviation are taken over 100 runs. The Pharo compiler takes source code as input and outputs Smalltalk bytecodes. Nabujito takes bytecodes as input and outputs native code.

We see that in the simple case displayed in Table 3 Nabujito's compilation speed lies within the same range as the standard Smalltalk compiler. We expect that in the future we apply more low-level optimizations and thus increase the compilation time of Nabujito. However we have shown in the performance evaluation for NativeBoost, the Benzo-based FFI, in Section 4.1.4 that even a rather high one-time overhead is quickly amortized. Furthermore with Smalltalk's image approach the generated native code is persistent over several sessions. A subsequent restart of the same runtime will not cause the JIT to nativize the same methods it did during the last launch. Hence our approach is even valid for short-timed script-like applications as most of the methods will already be available in optimized native code from a previous run.

4.3.5 Outlook

One major performance optimization missing in both, the original VM-level JIT and Nabujito, is inlining. By inlining we are able to create methods that are potentially big enough for optimizations. However inlining is a difficult task in a highly dynamic language such as Smalltalk. Efficient inlining can only be performed with sufficient knowledge of the system. Accessing this high-level information from within the VM is cumbersome and requires duplication of language-side reflective features. We are convinced that with Nabujito we simplify this task significantly. The JIT lives on the same level as the information it needs relying on the already present reflective features of Smalltalk.

5. Implementation Issues

The aspects concerning security for this kind of low-level capabilities over high-level languages allow for much discussion and controversy. Performance is the other most discussed is-

sue in the system programming domain and we exposed the results of Benzo related to it.

5.1 Security in Reflective Low-level Programming

Benzo breaks the security aspects provided in high-level languages such as memory safety or proper exception handling [23]. However the implications are not different from any other FFI implementation used in high-level languages. Direct use of low-level native instructions poses a security risk to the system. There has been detailed research in how to make FFI implementations more secure. Typically the compiler statically ensures that no compromising structures leave the VM-realm [18]. By analyzing the internal usage pattern of the external function it is possible to further reduce the risk of accidentally modifying vital internal VM structures. By shielding of the VM internal structures from the external world we effectively limit the risk but at the price of limiting also the power of an FFI. We show in Section 2 why FFI existent solutions are not powerful enough for certain types of extensions that are important for us. Security risks are one of the reasons exposed for this limitations.

Besides of the inherent security problems of FFI there is the whole reflective power of the Smalltalk environment as a security risk. Smalltalk allows a programmer to change classes and methods at runtime. There exist even methods that dynamically replace all references to an object with another one in the entire system. For many other high-level languages such functionality is not accessible from language-side or not present at all in the runtime. Some Smalltalk language features, such as the live instance migration [17], rely on this reflective capabilities and are vital for the developer experience. Hence we can conclude that Benzo poses the same security risks as other essential architectural decisions that the Smalltalk environment promotes.

We believe that security has to be addressed in a more general way at language-side and not restricting the possibilities of the developers. If we can enforce proper security constraints at language-side it is possible to encapsulate dangerous behavior in a controlled domain. Only with such a solution are we able to provide security in a Smalltalk-like environment.

5.2 Performance

Benzo allows the generation of efficient native code. We already showed that the generated native-code from language-side only causes a one-time overhead on its initial creation. Thereafter it is cached for later activations. We also argued that for the three Benzo proof of concepts examples proposed in Section 4 this overhead can be neglected. Furthermore, for the FFI implementation we show in Table 1 how we outperform the existing FFI implementations due to more specific native code. The performance gain by the execution of custom-made native code outweighs the one-time cost of language-side code generation.

Benzo's close interaction with the JIT described in Section 3.1.5 further reduces the reoccurring costs of calling native-code.

LuaJIT follows the same approach for their FFI library [1].

Our conclusion is that even a high one-time compilation overhead has little influence on the overall performance of the system. Hence the benefits of reflective low-level programming outweigh.

6. Related Work

QUICKTALK [7] follows a similar approach as Waterfall. However Ballard et al. focus mostly on the development of a complex compiler for a new Smalltalk dialect. Using type annotations QUICKTALK allows for statically typing methods. By inlining methods and eliminating the bytecode dispatch overhead by generating native code QUICKTALK outperforms interpreted bytecode methods. Compared to Waterfall QUICKTALK does not allow to leave the language-side environment and interact closely with the VM. Hence it is not possible to use QUICKTALK to modify essential primitives.

High-level low-level programming [16] encourage to use high-level languages for system programming. Frampton et al. present a low-level framework packaged as `org.vmmagic`, which is used as system interface for Jikes, an experimental Java VM. Additionally their framework is successfully used in MMTK [9] which is used independently in several other projects. The `org.vmmagic` package is much more elaborate than Benzo but it is tailored towards Java with static types. Methods have to be annotated to use low-level functionality. Additionally the strong separation between low-level code and runtime does not allow for reflective extensions of the runtime. Finally, they do not support the execution and not even generation of custom assembly code in the fly.

Other related approaches are VM generation frameworks in general. They try to abstract away the complexity of the VM and use high-level languages as compiler infrastructure. A very successful research project is Jikes Research VM [20]. It uses Java to metacircularly define a Java runtime which then generates the final VM. A similar framework is PyPy [29] a VM framework including an efficient JIT. PyPy uses a restricted subset of the Python language named RPython which is then translated to various low-level backends such as C or LLVM code. There exist several different high-level language VM implementations on top of PyPy such as Smalltalk [10] or Prolog. However its main focus lies on an efficient Python interpreter.

Other high-level languages such as Lua leverage FFI performance by using a close interaction with the JIT. LuaJIT [1] for instance is an efficient Lua implementation that inlines FFI calls directly into the JIT compiled code. Similar to Benzo this allows to minimize the constant overhead by generating custom-made native code. The LuaJIT runtime is mainly written in C which has clearly different semantics than Lua itself. Compared to our approach the efficient VM implementation suffers from the shortcomings described in Section 2.3.

Kell and Irwin [21] take a different look at interacting with external libraries. They advocate a Python VM that allows for dynamically shared objects with external libraries. It uses the low-level DWARF debugging information present in the external libraries to gather enough metadata to automatically generate FFIs. However they do not focus on the reflective interaction with low-level code and the resulting benefits.

7. Conclusions

We presented Benzo a reflective low-level programming framework written in a dynamic high-level language. Benzo is an integral approach for reflective high-level low-level programming. Using Benzo we efficiently implemented at language-side three distinct language feature extensions that typically reside at VM level.

Benzo promotes interaction with the low-level world by dynamically generating native code from language-side. This allows to exploit the underlying platform capabilities only when strongly needed without leaving the development platform and through a high-level programming interface. Benzo advocates the use of development tools and abstraction level of the high-level language for as much as possible or desired.

With high-level reflection capabilities combined with efficient low-level code we manage to do dynamic primitive instrumentation and reuse the code for primitive operations which is duplicated on the standard JIT approach. We also show that since Benzo caches native code transparently at language-side our JIT compiler poses only a one-time overhead when generating native code. Our mature FFI implementation outperforms an existing C-FFI implementation by a factor 1.5 even though we control every aspect from the language-side.

Benzo shows that promoting clear interfaces for controlling low-level code completely from language-side produces efficient solutions for system programming requirements without resorting to pure low-level solutions. We showed that combining the abstraction provided by high-level languages with the complete and precise powerful system programming capabilities of low-level languages is not only possible but profitable. Furthermore we manage to considerably reduce complexity and code duplication which results in better maintainability.

Acknowledgments

This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council, FEDER through the 'Contrat de Projets Etat Region (CPER) 2007-2013', the Cutter ANR project, ANR-10-BLAN-0219 and the MEALS Marie Curie Actions program FP7-PEOPLE-2011-IRSES. Also we would like to thank Marcus Denker, Damien Pollet and Ciprian Teodorov for kindly reviewing earlier drafts of our paper.

References

[1] LuaJIT FFI Library. `http://luajit.org/ext_ffi.html`.

[2] Waterfall. `http://lafhis.dc.uba.ar/waterfall`.

[3] B. Alpern, C. R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J. J. Barton, S. F. Hummel, J. C. Sheperd, and M. Mergen. Implementing Jalapeño in Java. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '99, pages 314–324, New York, NY, USA, 1999. ACM.

[4] A. Andersen. A note on reflection in Python 1.5. In *Lancaster University*, 1998.

[5] A. W. Appel. *Modern compiler implementation in Java*. Cambridge University Press, New York, NY, USA, 1998.

[6] J. Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113, June 2003.

[7] M. B. Ballard, D. Maier, and A. W. Brock. QUICKTALK: a Smalltalk-80 dialect for defining primitive methods. *SIGPLAN Not.*, 21(11):140–150, June 1986.

[8] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. *Pharo by Example*. Square Bracket Associates, 2009.

[9] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and water? High performance garbage collection in Java with MMTk. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 137–146, Washington, DC, USA, 2004. IEEE Computer Society.

[10] C. F. Bolz, A. Kuhn, A. Lienhard, N. D. Matsakis, O. Nierstrasz, L. Renggli, A. Rigo, and T. Verwaest. Back to the future in one week – implementing a Smalltalk VM in PyPy. *Self-Sustaining Systems*, pages 123–139, 2008.

[11] C. Bruni, L. Fabresse, S. Ducasse, and I. Stasenko. Language-side foreign function interfaces with nativeboost. In *Submitted to International Workshop on Smalltalk Technologies 2013*, 2013.

[12] G. Cooper. DTrace: Dynamic tracing in Oracle Solaris, Mac OS X, and free BSD by Brendan Gregg and Jim Mauro. *SIGSOFT Softw. Eng. Notes*, 37(1):34, Jan. 2012.

[13] M. Denker, J. Ressia, O. Greevy, and O. Nierstrasz. Modeling features at runtime. In *Proceedings of MODELS 2010 Part II*, volume 6395 of *LNCS*, pages 138–152. Springer-Verlag, Oct. 2010.

[14] K. Fisher, R. Pucella, and J. Reppy. Data-level interoperability. In *Electronic Notes in Theoretical Computer Science*, page 2001, 2000.

[15] D. Flanagan and Y. Matsumoto. *The Ruby programming language*. O'Reilly Media, Incorporated, 2008.

[16] D. Frampton, S. M. Blackburn, P. Cheng, R. J. Garner, D. Grove, Eliot, and S. I. Salishev. Demystifying magic: high-level low-level programming. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 81–90, New York, NY, USA, 2009. ACM.

[17] A. Goldberg and D. Robson. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Reading, Mass., May 1983.

[18] M. Hirzel and R. Grimm. Jeannie: granting Java native interface developers their wishes. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 19–38, New York, NY, USA, 2007. ACM.

[19] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: The story of Squeak, a practical Smalltalk written in itself. In *OOPSLA'97: Proceedings of the 12th International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 318–326. ACM Press, Nov. 1997.

[20] The Jikes research virtual machine. `http://jikesrvm.sourceforge.net/`.

[21] S. Kell and C. Irwin. Virtual machines should be invisible. In *VMIL '11: Proceedings of the 5th workshop on Virtual machines and intermediate languages for emerging modularization mechanisms*, page 6. ACM, 2011.

[22] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[23] S. Li and G. Tan. Finding bugs in exceptional situations of JNI programs. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 442–452, New York, NY, USA, 2009. ACM.

[24] P. Maes. Concepts and experiments in computational reflection. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 147–155, Dec. 1987.

[25] J. Malenfant, M. Jacques, and F. N. Demers. A tutorial on behavioral reflection and its implementation. Proceedings of the Reflection '96 Conference, 1996.

[26] E. Miranda. Context management in VisualWorks 5i, 1999.

[27] E. Miranda. The Cog Smalltalk virtual machine. In *VMIL '11: Proceedings of the 5th workshop on Virtual machines and intermediate languages for emerging modularization mechanisms*. ACM, 2011.

[28] J. Reppy and C. Song. Application-specific foreign-interface generation. In *Proceedings of the 5th international conference on Generative programming and component engineer-*

ing, GPCE '06, pages 49–58, New York, NY, USA, 2006. ACM.

[29] A. Rigo and S. Pedroni. PyPy's approach to virtual machine construction. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 944–953, New York, NY, USA, 2006. ACM.

[30] D. Ungar and R. B. Smith. Self. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, New York, NY, USA, 2007. ACM.

[31] D. Ungar, A. Spitz, and A. Ausch. Constructing a metacircular virtual machine in an exploratory programming environment. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 11–20, New York, NY, USA, 2005. ACM.

[32] T. Van Cutsem and M. S. Miller. Proxies: design principles for robust object-oriented intercession APIs. *SIGPLAN Not.*, 45:59–72, Oct. 2010.

[33] J. Vraný, J. Kurš, and C. Gittinger. Efficient method lookup customization for Smalltalk. In *Proceedings of the 50th international conference on Objects, Models, Components, Patterns*, TOOLS'12, pages 124–139, Berlin, Heidelberg, 2012. Springer-Verlag.

[34] C. Wimmer, M. Haupt, M. L. Van De Vanter, M. Jordan, L. Daynès, and D. Simon. Maxine: An approachable virtual machine for, and in, Java. *ACM Trans. Archit. Code Optim.*, 9(4):30:1–30:24, Jan. 2013.

Design and implementation of Bee Smalltalk Runtime

Javier Pimás
Disarmista S.R.L.
pocho@disarmista.com

Javier Burroni
Disarmista S.R.L
javier@disarmista.com

Gerardo Richarte
Disarmista S.R.L
gera@disarmista.com

Abstract

Bee is a Smalltalk dialect. Its runtime is exceptional in that it is completely written in Smalltalk. Bee includes a minimal kernel with on-demand loaded libraries, a JIT compiler, an FFI interface, an optimizing SSA-based compiler, a garbage collector, and native threading support among other things. Despite being written in Smalltalk, Bee achieves promising performance levels.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Code Generation; Compilers; Incremental compilers; Memory management (garbage collection); Run-time environments

General Terms

Keywords runtime, virtual machine, self-hosting, compiler, garbage collector

1. Introduction

Bee runtime is completely written in Smalltalk. This covers kernel features like message dispatching, primitives, just-in-time compiling, threading support and garbage collection, among others.

The implementation of this environment in such a high level language required solving key problems.

Insufficient meta-object semantics. Smalltalk includes a very powerful metacircular class hierarchy. Yet it doesn't reify a key aspect of objects: it is not possible to access object headers. This poses barriers to the implementation of things like primitives, or garbage collectors. We slightly augment the smalltalk semantics by implementing *underprimitives*, extremely small and efficient methods that consist of just a few inline-assembled machine instructions.

Breaking self-sustaining circularity. Bee is self-hosted, which means it doesn't require any external runtime support library or virtual machine. Features like message lookup, primitives and garbage collection are implemented within the language itself. This characteristic means that the code implementing some features often assumes and even requires their existence to work. As an example, message lookup is written in Smalltalk, therefore message lookup naturally sends messages during its own execution. This leads to endless lookup recursion unless a means to cut it is incorporated. We worked around this kind of problems by carefully dissecting *code closures* and by issuing ahead-of-time nativization[1] of dispatch mechanisms.

A key aspect to support our solution is having control of the Smalltalk JIT compiler and machine code assembler, which are both written in Smalltalk. The JIT lets us transform underprimitives into very low level and efficient pieces of code, leveraging low-level actions in an object oriented fashion.

The remaining of this paper is organized as follows. In section 2 we describe the context of this Smalltalk dialect. In section 3 we describe the three most relevant models for the Bee runtime: the Bee metaclass hierarchy, the memory model and the ABI model. The runtime implementation details are described in section 4. We focus on the implementation of method lookup and primitives, but we also describe the modularity in the design, and the mechanism to perform low-level operations from Smalltalk, id est., the underprimitives. Current state of Bee development is described in section 5. An analysis of performance can be found in section 6. We culminate this work with a discussion on related work in section 7 and a conclusion with final remarks in section 8.

[1] We use the term nativization to mean generation of machine instructions. This is to contrast with the word compilation, which we use to refer to bytecode generation; or the word jitting, which we use for just-in-time nativization

2. Context

High level languages usually allow a more dynamic programming style by delaying bytecode and machine code compilation, avoiding static typing and adding automatic garbage collection, among other things.

High level languages require *runtime support* to offer all these functionalities, usually in the form of a Virtual Machine. These functionalities have runtime costs that drag down performance of user programs. Besides, higher level languages discourage or even disallow low level actions like direct access to memory for the sake of program safety. Even if possible, accesses to memory are done through abstractions that try to validate each action, heavily hurting performance.

This combination of characteristics makes it difficult to implement runtime support itself in high-level languages, resorting instead to low-level ones.

Low level languages, on the other hand, usually require static compilation to machine instructions before execution, type specifications throughout the code and manual memory management. In exchange of this, low level languages usually generate highly efficient code.

But implementing runtime support libraries in high level environments can yield a better understanding of the problem's domain [20]. Runtime programmers can take advantage of the environment tools and abstractions. Instead of spending their focus simulating code execution in their heads, they can make use of the plethora of inspectors, browsers and debuggers to give shape to more readable and easier to understand solutions.

Finally, programmers want and should be able to know, understand and improve the implications and limitations of the runtime they are running on [14]. This is eased if the programming language of the runtime is the same than the language they use everyday for writing code.

3. Overview of Bee metaclass and memory model

Before delving into complex Bee topics like machine code generation, lookup and primitives implementation, we give an overview of a small group of Bee details that will help understanding the whole system.

3.1 Bee metaclass hierarchy

Bee follows as a base Smalltalk-80 class hierarchy [9], with some major deviations. The root class in the hierarchy is ProtoObject, whose super class is nil; Object subclassifies ProtoObject.

A big difference between Bee and Smalltalk-80 lays in its metamodel. The metaclass hierarchy, while similar, has been severed to allow dissociating class shape and object protocol. This simplifies the usage of objects of a same class with different behavior. The class Behavior truly refers to object behavior. It is not the superclass of Class and Metaclass. Instead, it is a variable size collection of method dictionaries. During lookup, the object's behavior will be traversed in order until a method is found for the searched selector. Thus, an object's behavior defines how the object responds to messages.

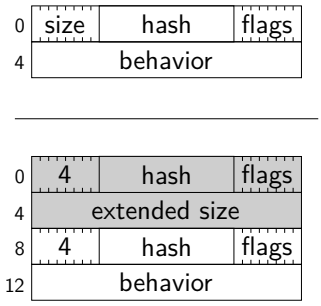


Figure 1. Regular and extended object headers

Both Class and Metaclass subclassify Species. Species includes most of Smalltalk-80 protocol for ClassDescription, which handles class and instance variable names, instance creation and more.

3.2 Memory model

In memory, objects are stored as an array of slots or bytes (depending on whether they are pointer o byte objects) preceded by a header. Slots that don't reference SmallIntegers have the address of the first slot or byte of the referenced objects, just after the header. We call these OOPs (Object Oriented pointers).

Given an object, its header describes many of its properties, like size, hash, shape and behavior. It is composed of various bit fields. It occupies 8 bytes in regular cases, or 16 if it is an extended header. Extended headers are needed for big objects (bigger than 255 elements) and for ephemerons [10]. The first doubleword of a regular header describes size, hash, shape and garbage collection status. The second doubleword is an OOP to the object's Behavior. When extended, the header contains two additional doublewords that are placed immediately before the regular header. The size field of regular header is set to 4 and the isExtended bit is turned on. The first doubleword of the extended header is set as a copy of the first doubleword of the regular header. The second word, on the other hand, is set to the actual size of the object. The size of the object represents the number of slots or bytes in memory of the object.

Notice that objects don't have a direct pointer to their class. Their class is determined by the class field found in the first method dictionary of their Behavior. Figure 1 shows the memory layout of object headers.

SmallIntegers are an exception to these memory layout rules, they are tagged. SmallIntegers are not allocated in the heap. When a slot is stored with a SmallInteger, instead of writing a memory address, we write the numeric value shifted one bit to the left and incremented by one. As objects are aligned in memory to 4 bytes addresses, SmallIntegers can be quickly distinguished from regular objects. This is a common technique that was already present in the 16-bit implementation of Smalltalk-78 [15], and adopted by many other virtual machine implementations later [6, 13].

We extend SmallInteger tagging with the use of *small pointers*. SmallIntegers represent numbers from -2^{30} to $2^{30}-1$, as they fit in a 32-bit word with the least significant bit clamped at 1. When dealing with pointers, we use standard SmallIntegers to do arithmetic calculations. This limits us to pointers with addresses in the 0 to $2^{30}-1$ range, exactly 1GB of memory.

Conversion of pointers to SmallIntegers is done by shifting the pointer to the left 1 bit and adding 1 to the result. But as pointers are always 4 byte aligned, we can convert them to SmallIntegers by just setting their least significant bit to 1, without shifting. The small integer represented by such a doubleword is the memory address divided by two. We call this a small pointer. Small pointers look and behave exactly as SmallIntegers, the programmer is responsible of dealing with conversions when needed. Thanks to small pointers, we are able to support up to 2GB of memory.

3.3 Bee ABI[2]

Bee assembler models a Stack architecture with a group of very specific registers, as described in [1]. These are: R (receiver and result), arg (argument), temp, self, method environment context, stack base and top of the stack. Both R, arg and temp can change between bytecode and bytecode, and are saved by the caller during message send. Self doesn't change from bytecode to bytecode but must be restored before returning from a method. Arguments are passed in the stack, pushed left to right, and are callee cleaned.

Currently, Bee only supports x86-32 bit architecture, and we map R to EAX, arg to EDX, temp to ECX, self to ESI, method environment context to EDI, frame pointer to EBP and top of the stack to ESP[3]. We show a stack frame for this ABI in figure 2. In the example, the method receives two arguments, contains two temporary variables and at least one block closure.

Most Smalltalk methods generate a new stack frame on activation, unless they are very short and don't need one. After pushing the previous frame pointer, they push the receiver and the compiled method. If necessary, they also push the method environment context. On exit, the stack top is set to the frame pointer, the old frame pointer is popped and ret n instruction transfers control back to the caller cleaning n arguments.

[2] The application binary interface defines low level conventions like parameter passing and saved registers across calls

[3] this convention is very similar to Pascal

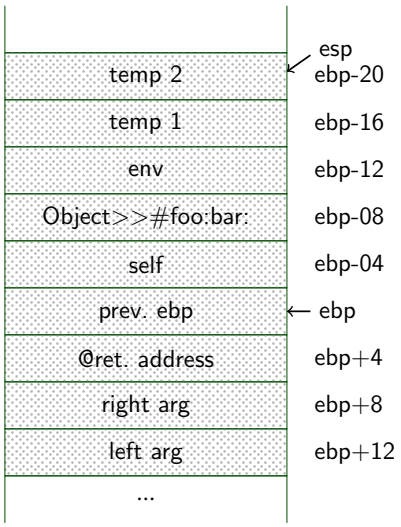


Figure 2. Bee Stack frame layout for Object>>#foo:bar: in memory on x86

4. Bee implementation details

4.1 Bee method nativizer

All methods in Bee are objects of the class CompiledMethod. Methods are compiled to bytecodes first, and then nativized. Among other things, CompiledMethods contain slots pointing to their bytecodes and to their NativeCode, a reification of machine code. NativeCode, in turn, contains a slot that points to the actual machine code, which is stored as a ByteArray. NativeCode also contains an array of references, with their respective offsets inside the machine code ByteArray. In this way, machine code is abstracted, can be easily accessed by the image when needed (i.e. during lookup), and needs little special treatment by the runtime, as shall be seen later.

To generate machine code from a method's bytecodes, an instance of the class BeeMethodNativizer iterates over the bytecodes writing their corresponding assembly.

As in Smalltalk-80, Bee Smalltalk contains special bytecodes for the most common arithmetic and logic operations. Our compiler is also smart enough to transform simple blocks, such as the ones used in #ifTrue:, #ifFalse: or #whileTrue: messages, into equivalent comparison and jump bytecodes. Besides, Bee is also capable of doing special case message send nativization for selectors that are not associated with a special bytecode. We detail both approaches next.

4.1.1 Inline nativization of messages through special bytecodes

The method nativizer generates specific assembly for most cases of special arithmetic and logic selectors. As example, let's consider addition. At compile time, when encountering a #+ message send, the compiler will output a Plus bytecode. At nativization time, the nativizer will assemble a couple of instructions to perform inline addition if possible, and to send the #+ if not. First it will insert a SmallInteger test for both the receiver and the argument. Then it will assemble the addition and a check if the result fits in a SmallInteger. Finally it will assemble the #+ message send. At runtime, if all checks pass, the #+ send will be skipped; if any of the mentioned checks fail, it will fallback to the message send. Even if this is mainly done for performance reasons, it has a deep impact in other parts of the system, easing the implementation of components like lookup and the garbage collector.

4.1.2 Custom nativization of message sends through send inliners

When the nativizer passes through a generic message send bytecode, it delegates the machine code generation to different assembly generators, or *send inliner*s. The method nativizer associates selectors with different send inliners. In the typical case, the associated send inliner will assemble the necessary instructions to call lookup. For some specific selectors, on the other hand, the nativizer will associate a different send inliner and generate different assembly.

Being in control of the bytecode nativizer from Smalltalk is critical for the implementation of these different send inliners, which are essential to leverage the development of Bee runtime in an efficient and object oriented manner. They are also key for system self-sustainability.

Different send inliners include the *assembly send inliner*, which generates special case machine code depending on the selector, and both *lookup send inliner* and *invoke send inliner* which generate machine code to call lookup and invoke respectively.

Underprimitives. These are a minimal set of selectors that are resolved with inline assembly, instead of sending a message. An example of an underprimitive is #_isSmallInteger. When seeing this selector, the assembly send inliner directly inserts assembly to check if the object is tagged.

```
assembleTestSmallInteger
  | integer |
  integer := assembler testAndJumpIfInteger.
  self loadObject: false.
  assembler uncontionalSkip: [
    assembler jumpDestinationFor: integer.
    self loadObject: true]
```

Underprimitives are a convenient abstraction of very low-level operations. They are limited to a maximum of two arguments. They assume the receiver is in R register and that the first and second arguments lay in arg and temp respectively, if present. A few dozen of underprimitives are enough to cover all the low-level actions needed for the implementation of the entire system.

4.1.3 Lookup and invoke

Execution of Smalltalk code involves execution of a message dispatching algorithm. In Bee, this algorithm is written in Smalltalk. For that reason, it is necessary to cut the recursive lookup chain to avoid an infinite recursion.

We shall distinguish two mechanisms when issuing what is generically called lookup: *method lookup* and *method invocation*. The first refers to the action of finding the corresponding compiled method to be later executed. The second one refers to the action of transferring control to the compiled method's native code.

In the next snippet we show the #_lookupAndInvoke entry method. The code is straightforward: #_lookup: fetches the corresponding compiled method for the selector, or nil if none was found, in which case the message to send is #doesNotUnderstand:. The compiled method is prepared for execution and finally control is transferred to the found method's native code. Notice that Bee uses *monomorphic inline caches* [8], so lookup includes call-site patching code.

```
_lookupAndInvoke: aSymbol
  | cm |
  cm := self _lookup: aSymbol.
  cm == nil ifTrue: [
    cm := self _lookup: #doesNotUnderstand:.
    self _transferControlTo:
          cm noClassCheckEntrypoint _asNative].
  cm prepareForExecution; patchClassCheckTo: self behavior.
  self
    _transferControlDiscardingLastArgAndPatchingTo:
              cm noClassCheckEntrypoint _asNative
```

Lookup. The native code generator used for lookup is the same than the one used for any other Smalltalk methods, with a slightly different configuration for message sends. When nativizing the #_lookup: message send, it will generate machine code according to the configured send inliner. If using the lookup send inliner, this code would fall into an infinite recursion. To solve this problem, we calculate a *code closure*. The implementation of #_lookup: is unique to all system, and we know beforehand the compiled method that would be found if #_lookup: were looked up. We can assure, by carefully writing lookup code, that the same happens to all the messages involved in lookup. Then, when nativizing lookup methods, we can set the send inliner to an *invoke send inliner*. An invoke send inliner pushes the unique compiled method for that selector, instead of pushing a generic

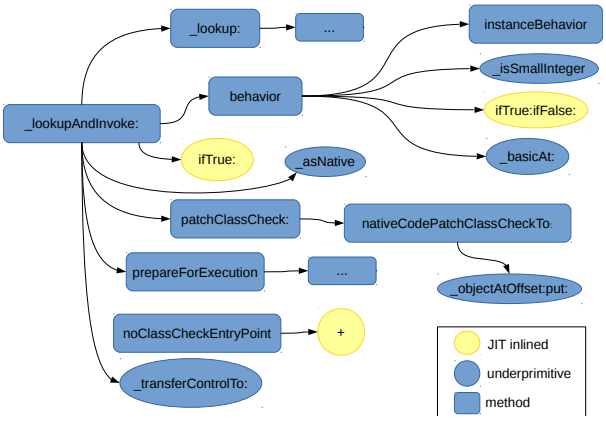


Figure 3. Code closure of #lookupAndInvoke:. Some selectors were summarized for brevity.

selector, and calls an invoke mechanism, instead of lookup. Figure 3 shows the message send graph of #lookupAndInvoke:, for which the method closure is calculated. We manually configure the method nativizer send inliners, so that message sends get nativized as invokes to the methods to be found for the respective selectors.

Invoke. We detail invoke implementation next. It is very similar to the end of #lookupAndInvoke:. The main difference is that invoke patches the call site to point to the native code just after the prologue. This avoids a class check in the next call (which would fail if the next receiver is of a different class).

```
_invoke: aCompiledMethod
  aCompiledMethod prepareForExecution.
  self _transferControlDiscardingLastArgAndPatchingDirectTo:
    aCompiledMethod noClassCheckEntrypoint _asNative
```

There is a last subtlety with invoke. Consider the nativization of #prepareForExecution message send. Of course, if it were nativized with a lookup send inliner, it would cause an infinite recursion at runtime. But if nativized with the very same invoke send inliner, it would also cause an infinite recursion of invokes. The trick to solve this is to nativize #_invoke: with a shortened and inlined version of itself. The following snippet shows the resulting code, where all message sends are actually nativized inline, and can be resolved without sending any real message.

```
_lightweightInvoke: aCompiledMethod
  | nativecode bytes classCheckDisplacement address |
  nativecode := aCompiledMethod _basicAt: 2.
  bytes := nativecode _basicAt: 1.
  classCheckDisplacement := 16rD.
  address := bytes _asSmallInteger +
    classCheckDisplacement.
  self
    _transferControlDiscardingLastArgAndPatchingDirectTo:
      address _asNative
```

To finish this section we explain why _lookupAndInvoke: can be a simple Smalltalk method with normal arguments, and how the control transferring works.

Arguments passing. As can be derived from section 3.3, just before the call to method lookup, all method arguments have been pushed in left-to-right order, and lastly, the selector was pushed and lays in the top of the stack. #_lookupAndInvoke: takes advantage of this fact. Inside #_lookupAndInvoke:, the references to the first and only argument will be transformed by the nativizer to the same address it would do in any case: ESP+8. This works perfectly until the return point.

Transefering control. At the epilogue of #_lookupAndInvoke: and #_invoke:, just after restoring frame pointer and stack top, the stack still has an extra argument, the selector or compiled method, respectively, that needs to be removed. If not, the method to be activated would wrongly see the selector as it rightmost argument. There is an extra complication, because after the selector was pushed, the call to lookup was issued, and the return address was pushed into the stack. _transferControlTo family of underprimitives write the same assembly that is commonly issued on method exit, but also solve this last argument problem by issuing pop [esp], an instruction that pops the top of the stack into its next position. Besides, as last instruction, it assembles a jmp instruction, instead of a ret n, seamlessly transferring control to the actual method, which doesn't need any special stack treatment.

4.2 Access to object headers

Thanks to underprimitives, it is possible to access raw object headers. Yet, these underprimitives are a bit too low-level for common usage. As an example, checking the size of an object requires reading the header bits to determine if it is extended, and then to access the corresponding size field, which might be a byte or a whole slot. Implementing all this with an underprimitive would be overkill, as it would require too much assembly writing. Instead of that, we implemented a set of methods that abstract access to object headers within Smalltalk code. For example, checking if an object is extended can be done with the following line of code:

```
_isExtended
  ^(self _basicFlags bitAnd: IsExtended) = IsExtended
```

Again, these are methods written in Smalltalk, so we can easily do complex actions. An object header can be marked as bytes with:

```
_beBytes
  self _flagsSet: IsBytes.
  self _isExtended ifTrue: [self _extendedFlagsSet: IsBytes]

_extendedFlagsSet: mask
  self _extendedFlags: (self _extendedFlags bitOr: mask)
```

The size of an object can be obtained from its header with:

```
_size
  | total |
  total := self _isExtended
    ifTrue: [self _extendedSize]
    ifFalse: [self _basicSize].
  ^(self _isBytes and: [self _isZeroTerminated])
    ifTrue: [total - 1]
    ifFalse: [total]
```

These methods abstract away most problems of dealing with object headers in a clean, object oriented style. A discussion about their efficiency is done in section 6.3.

4.3 Primitives

Bee doesn't implement primitives in the standard Smalltalk-80 way. The reason for this are undermethods, underprimitives and inline nativization of bytecodes. We begin with the description of the simplest primitives, and finish the section with the most complex ones, showing how most of the code can be implemented in plain Smalltalk, with the help of only a few underprimitives.

It is also important to remark that in current Bee iteration garbage collection has not yet been enabled. This eases implementation of primitives but will need revision when garbage collection is enabled again.

We start explaining this by showing a very small example. Consider the method ProtoObject>>#size. While in other Smalltalks this will need a primitive, in Bee it will be implemented as:

```
ProtoObject >> #size
  ^self _size
```

The implementation takes advantage of the reification of the object header, which can be accessed through undermethods. Other good examples are ProtoObject>>#behavior and ProtoObject>>#class

```
ProtoObject >> #behavior
  ^self _isSmallInteger
    ifTrue: [SmallInteger instanceBehavior]
    ifFalse: [self _basicAt: 0]
```

```
ProtoObject >> #class
  ^self behavior mainClass
```

Notice that Behavior has been reified, so finding the class can be delegated to it. ProtoObject>>#== shows the benefits of inline nativization of bytecodes:

```
ProtoObject>>#== other
  ^self == other
```

This will be nativized as a pointer comparison by the nativizer. If the pointers are equal it will load true, else it will load false. ProtoObject>>#perform: is a good example of the benefits of the reification of lookup.

```
ProtoObject>>#perform: aSymbol
  aSymbol arity = 0 ifFalse: [^self error: 'incorrect arity'].
  ^self lookupAndInvoke: aSymbol
```

Unlike Smalltalk-80 primitives, here there is no special concept of primitive failure. When a wrong arity is detected in normal Smalltalk code and doesn't require a second chance method activation. Careful readers will notice that the sent message is #lookupAndInvoke: and not #_lookupAndInvoke:. The difference is that the former doesn't patch the call site during invocation, which would be wrong in the case of perform. The main advantage of implementing low-level functionality in Smalltalk is that we can rely on existing code. For example, calculating selector arity was already implemented code. This gets an even bigger impact when writing more complex primitives. Consider the implementation of #value

```
BlockClosure>>#value
  self argumentCount = 0 ifFalse: [^self arityError].
  self _transferControlTo: self code
```

Notice how natural this code feels. code returns the address of the block's native code. The only addition to Smalltalk semantics needed was the #_transferControlTo: underprimitive. Other variations with a different amount of arguments are very similar.

The implementation of #become: is very interesting. Remember that #become: should scan all Smalltalk memory looking for references to the receiver, and replacing them with the argument. Besides, the process' stack, which is not inside a GCSpace, should also be visited.

```
ProtoObject>>#become: anotherObject
  Memory current make: self become: anotherObject

Memory>>#make: anObject become: anotherObject
  1 to: spaces size do: [:i | | space |
    space := spaces at: i.
    space make: anObject become: anotherObject].
  ProcessStack current make: anObject become:
    anotherObject
```

#become: is split in two stages. The first stage traverses each existing GCSpace, looking for references to the source object, and replacing them with the target one.

```
GCSpace>>#make: anObject become: anotherObject
  | objectBase object endOop |
  objectBase := self base.
  endOop := self nextFree.
  [objectBase < endOop] whileTrue: [
    object := (objectBase + 8 _asPointer) _asObject.
    object _isExtended
      ifTrue: [
        objectBase := (object _basicSize * 4)
                        _asPointer + objectBase.
        object := objectBase _asObject]
      ifFalse: [
        objectBase := objectBase + 8 _asPointer].
    objectBase := objectBase + object
                        _sizeInBytes _asPointer.
    0 to: object _pointersSize - 1 do: [:i |
      (object _basicAt: i) == anObject
        ifTrue: [object _basicAt: i put: anotherObject]]]
```

After all spaces have been scanned, the stack is traversed to find any remaining slot to change.

```
ProcessStack>>#make: anObject become: anotherObject
  | frame size endMarker nextFrame |
  frame := self _framePointer.
  endMarker := 0 _asObject.
  [
    nextFrame := frame _basicAt: 1.
    nextFrame == endMarker]
    whileFalse: [| first |
      size := nextFrame _asPointer -
                frame _asPointer // 4 _asPointer.
      first := 3.
      self
        make: anObject
        become: anotherObject
        in: frame
        count: size
        startingAt: first.
      frame := nextFrame]
```

GCSpace traversing has an extra subtle difficulty. The implementation avoids using real block closures. Real block closures require an environment, which is nothing more than an array to be allocated in the current GCSpace. This array might reference the source object and get modified during scan. If this happens, the source object might not be recognised any more. Besides #ifTrue:ifFalse: family of messages, both #whileTrue:, #whileFalse: and #on:do: are also inlined by the Smalltalk compiler. For example, when finding a #whileTrue: message send, the compiler inserts a jump-false bytecode at the block guard site, targeting the code after the argument block. It also inserts an unconditional back-jump at the end of the argument block, targeting the beginning of the guard block.

We close this section by showing Bee implementation of the most complex primitives, those related to blocks. We shall first give an overview of block mechanisms in Bee. We already showed the implementation of #value. Here we focus in the most difficult to implement piece, which is related to #ensure:. Consider the code

```
workSafe: aBlock
  [ aBlock value ] ensure: [ resource free ].
```

The meaning of this method is that after aBlock value is run, #free must also be run, no matter what happens in the block. To better understand the problem we can think how the stack looks like just after aBlock value, and how it will evolve. Somebody has called #workSafe: passing a block. To give a view of some of the different possibilities let's just assume it was:

```
sendWorkSafe
  ^self workSafe: [ a == b ifTrue: [^self] ]
```

In the stack we have #sendWorkSafe: frame, followed by #workSafe: frame. Next will be #ensure: frame. We can ignore what it does for now and assume that after a few extra frames aBlock frame will be placed in the top of the stack. The complete stack is shown in figure 4. Now, if a equals b, the ifTrue: branch will be executed, returning from #sendWorkSafe method. In stack frame terms, this means that stack should be unwound until #sendWorkSafe frame is found, and finally that frame should also be popped, returning control to #sendWorkSafe sender. But as there is an ensure in between, unwinding should pause when reaching the ensure stack frame, the ensured block should be executed and only after that unwinding should be continued. In the case that a was not equal to b, aBlock should finish its execution normally, and control should flow back to #ensure: normally, where it will activate the ensured block and return.

To allow the first case, #ensure: marks the stack to indicate an unwind stop point. In the case of premature return from a block, the stack will be traversed to find the returning

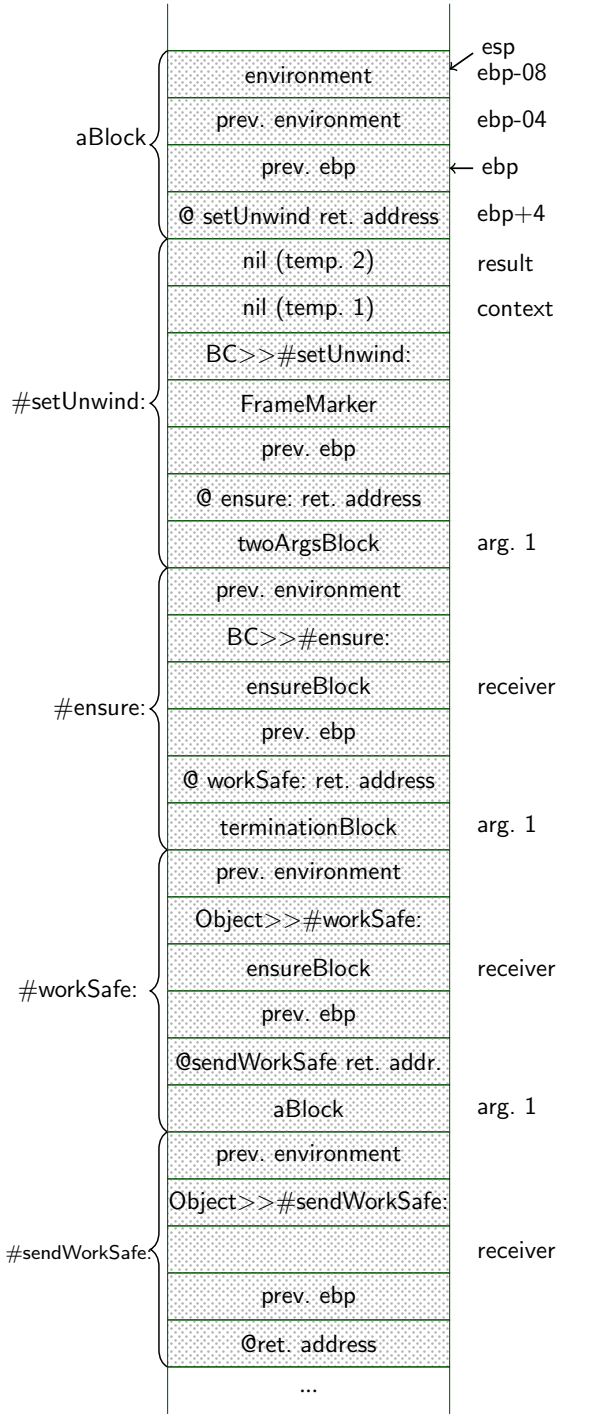


Figure 4. Stack frame layout after aBlock activation

method's frame. If a marker is found before that unwinding will stop. Here we show the code run when returning from a block.

```
BlockClosure>>#return: result
  | home environment saved frame |
  home := self methodEnvironment.
  home == nil ifTrue: [^self sendCantReturn].
  frame := BeeFrame current.
  [
    frame moveNext.
    frame isZero ifTrue: [^self sendCantReturn].
    frame hasBlock ifTrue: [
      saved := frame savedEnvironment.
      environment := frame environment].
    frame hasMarker
      ifTrue: [^self unwindUntil: frame context: saved
     returning: result].
    environment == home]
    whileFalse: [].
  saved == nil ifFalse: [saved _restore].
  ^frame current
    _dropUpperContextsReturning: result
    popping: self method argumentCount _asNative
```

a BeeFrame is a reusable stack frame reification. When initialized to current, it points to the top of the stack. It can be moved next to point to the next frame, and eases a lot the manipulation of stack frames. The #return: method, unwinds frame by frame looking for a marked frame or the returned method's frame. In the latter case it will drop all frames up to that point.[4] In the former, it will unwind just until the marker:

```
BlockClosure>>#unwindUntil: frame context: context
      returning: result
  frame firstTemporary: self; receiver: frame nextInFrame
      receiver.
  context == nil ifFalse: [context _restore].
  frame previous _dropUpperContextsReturning: result
      popping: 0 _asNative
```

The underprimitive used to drop stack frames is the same. This last method also does two modifications to the frame to be activated: changing its receiver and its first temporary. To explain why this is needed, we first show how the stack is marked:

[4] When walking the stack it will eventually find the method's frame, which has pushed an environment that is the same than the block's home

```
BlockClosure>>#setUnwind: twoArgumentBlock
  | context result |
  result := self valueMarked.
  context == nil ifFalse: [twoArgumentBlock value: context
    value: result].
  ^result

BlockClosure>>#valueMarked
  | receiver frame |
  self argumentCount = 0 ifFalse: [^self arityError].
  frame := BeeFrame current moveNext.
  receiver := frame receiver.
  frame receiver: FrameMarker.
  receiver _transferControlTo: self code
```

#setUnwind: method marks the stack. Notice that context, the first temporary is never directly assigned but checked for nil. Now remember the previous snippet of code, when unwinding to the marked stack frame. That code sets the first temporary of the frame, effectively making it not nil. Then, if context is not nil, it means the stack was unwound. If nil, there wasn't any non local return and the result is returned. The two argument block is then a block that is executed on marked stack unwinding. #ensure: uses it accordingly to guarantee that the ensured block is always executed.

```
BlockClosure>>#ensure: terminationBlock
  | result |
  result := self setUnwind: [:context :return |
    terminationBlock value.
    context return: return].
  terminationBlock value.
  ^result
```

Finally, #valueMarked code is similar to #value, but it takes the receiver and overwrites it with the marker. This is fine, as #valueMarked receiver can be restored from the previous stack frame (it is always sent by the same block).

4.4 Modularity

The strongest design principle behind Bee is minimality. Every aspect is split into smaller pieces as much as possible. Bee is divided into a very small kernel library, and a set of other libraries that can be loaded at runtime. Bee is self-hosted. This means it doesn't need to run on top of a Virtual Machine, all its runtime support is written in Smalltalk.

Bee libraries. Bee code is distributed through Smalltalk libraries, which are binary files that contain objects, including compiled methods and their native code.

Libraries are implemented as a heap of objects preceded by a description of the heap. To make loading fast, objects are stored almost as they will lay out in memory after loaded. The kernel includes a library loader, so it knows how to take the objects out of the library and how to plug them to the system. New classes are added to the Smalltalk global. New methods of already existing classes are inserted into their respective method dictionaries. All the needed actions are carried out to ensure that after loading the system stays consistent. Because of being binary based, library load time is small, compared to the time needed for compilation and nativization.

Bee kernel. Bee pushes Smalltalk modularity to new limits. Its kernel doesn't include a Smalltalk compiler, a nativizer, or a garbage collector. All of these functionalities are *optional*, and can be quickly loaded at runtime through libraries.

Bee is distributed as a native executable file. This file contains inside a kernel library with the main Smalltalk objects and code. The kernel library format is the same than the one used for any other library. The only difference is that it is packed inside a windows PE executable and that it contains no references to external objects. This kernel includes the minimal objects needed to be self-hosted. Main classes are placed in kernel, with their main methods. Methods contain their already nativized machine code. This is key for self-hosting. The entrypoint of the PE file is set to point to the machine code of a bootstrapping method. When execution starts, this method performs a basic initialization and then looks at the command line arguments to know what to execute next.

In Bee libraries, methods can be stored with or without their native code. Bee compiler and nativizer are placed in separate libraries, not included in the kernel and loaded on demand. Methods of libraries that include native code can be directly executed without loading the nativizer library. Libraries that don't include native code require loading the nativizer. When the nativizer is plugged, attempts to execute methods that don't contain native code automatically trigger their nativization. Of course, the methods of the nativizer library must be stored with their native code, as the native code of the nativizer is required to nativize methods. If both compiler and nativizer libraries are loaded, Bee will be able to execute arbitrary strings of Smalltalk code. This kind of modularity gives place to interesting possibilities. It is possible to create minimal system that is dynamic and yet doesn't include a compiler nor a nativizer within itself. To allow dynamism, the system could allow remote injection of compiled methods with their native code into the system from the outside world. This may prove useful for hardware platforms were resources are scarce.

5. Current Bee development

Bee is implemented on top of another *host* Smalltalk. This strategy lets us do development within a full blown environment. Many pieces of the system can be developed and tested in this environment. For example, Bee nativizer can be configured to generate machine code compatible with the

host environment. This allows testing most, if not all the nativizer functionality within the host.

In cases where testing within the host is not possible, we still can write the code inside the environment, and generate an executable file containing the kernel image and a library with the tests. Testing is conducted from the host. From it we spawn a Bee process, specifying the name of the test library as a command line argument. Test libraries are constructed to return value of 0 when the tests fail, or 1 if they succeed. Dynamism is transcendent, as changes done in the host environment can usually be tested immediately. Some other changes require the regeneration of the test libraries, which happens in just a few seconds. Only from time to time a change requires writing the kernel bootstrap image, because the system is split in libraries. Even in that case the time required is small, a few dozen of seconds.

As of June 2014, we are not yet able to directly debug Bee when running on itself. When this is needed, we resort to native code debuggers and disassemblers. For typical Smalltalk code, this will be solved after we plug the host's Smalltalk debugger and inspectors. Yet, a Smalltalk-written native code debugger would also be helpful to debug low-level code in a high-level environment.

In the previous iteration of Bee, a handful of garbage collectors were implemented. This includes full space mark and compact, and generational garbage collection. Yet, in the current form of Bee, we haven't finished plugging these collectors to the system. Therefore, there is no garbage collection available at all, until we adapt the old collectors.

6. Performance

Bee has been written with functionality and code quality as main priorities. Even though we haven't focused in performance yet, we still did implement some optimizations to obtain good enough performance for development. The philosophy has been to design the system with no inherent inefficient features, but to leave optimizations for later stages. The flexibility of the system facilitates research in this area.

6.1 Lookup optimizations

Bee is not interpreted, but ahead and just in time nativized. Besides, it utilizes monomorphic inline caches and different send inliners to enable fast dispatch. Assembly send inliners allow fast access to object headers, through directly writing machine code. Invoke send inliners provide for message sends without lookup, which is needed for lookup. We have taken advantage of this and configured the method nativizer to always use invoke for a set of very frequently sent messages. Through careful profiling and benchmarking we were able to remove the biggest performance bottlenecks.

The naïve #_lookupAndInvoke: method that was shown in section 4.1.3 was improved with a global lookup cache that speeds up lookup in the cases where mononomorphic inline cache fails. Currently we know that lookup is still a bottleneck, and we are working on the implementation of different optimizations to boost performance. When global cache fails standard lookup is done. Standard lookup is extremely slow, because it performs a linear scan in the method dictionaries of the object's behavior.

6.2 Optimizing compiler

The code generated by the JIT compiler is very efficient. However, to boost performance further, hot code paths could and should be made even more efficient. There is abundant research in this area that guarantees that important speed ups can be obtained. Adaptive optimization has been deeply studied, specially on Self [5, 11, 12, 19].

We have implemented an optimizing compiler. This compiler is run only for sets of methods that are known to be important performance-wise (for now, methods are selected manually). The optimizing compiler starts from an abstract syntax tree to construct an SSA-based call-flow graph of intermediate instructions [16, 18]. Through many stages it transforms this intermediate representation to finally emit native code. The different stages include speculative method inlining, peephole optimization, register allocation to finish in machine code emission. While still in early stages, this compiler has already provided a noticeable boost in performance.

6.3 Benchmarks

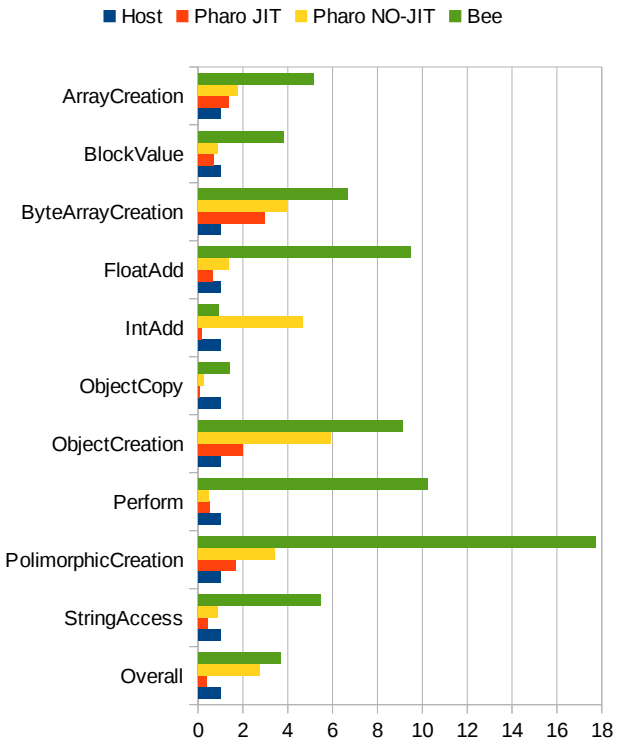
To measure performance we have run two different sets of benchmarks an compared the results against two other Smalltalk implementations: the host Smalltalk and Pharo [4]. In the case of Pharo, we have both run benchmarks with and without the JIT compiler.

The benchmark set is small but gives a view of the current Bee efficiency and also a preview of feasible performance levels that can be expected in Bee.

Slopstone is a well known Smalltalk benchmark that measures low-level operations as integer addition, block activation, object creation, and others. We split the results to give a better overview and also added some new sub-benchmarks to inquire about specific performance bottlenecks. Integer and float addition were tuned to run more iterations than in default Sloptone, because their execution time was so small that could not be correctly measured.

Some low-level performance details come to light in Figure 6.3. Results in this benchmark are highly diverse. On many cases Bee is between 3X and 6X slower than the host environment, with some notable exceptions. On the bright side, inline jitting makes integer addition even faster than the host virtual machine. On the other hand there are some notorious bottlenecks present on float operations, perform, monomorphic object creation and polymorphic object creation[5]. This last case is extremely slow because the

[5] With this we refer to creating objects of different classes



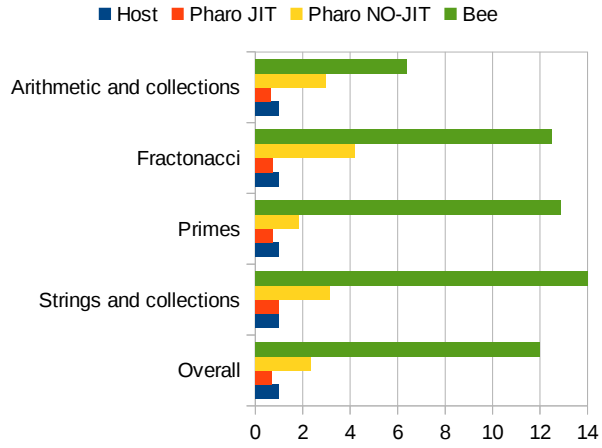
Benchmark	Pharo JIT	Pharo NO-JIT	Bee
ArrayCreation	1.36	1.77	5.16
BlockValue	0.71	0.89	3.8
ByteArrayCreation	2.98	3.98	6.67
FloatAdd	0.66	1.34	9.46
IntAdd	0.16	4.68	0.89
ObjectCopy	0.07	0.23	1.4
ObjectCreation	1.96	5.89	9.11
Perform	0.53	0.46	10.22
PolimorphicCreation	1.7	3.39	17.72
StringAccess	0.44	0.88	5.44
Overall	0.39	2.74	3.68

Figure 5. Normalized Slopstone execution times , relative to host virtual machine (lower is better).

monomorphic inline cache is not able to bear efficiently with polymorphic message sends.

Smopstone measures medium-level Smalltalk operations, which include recursive block and method calls, collection building and enumeration, streaming, and sorting. In a lower level, it performs arithmetic operations (mostly integer, with some fractions and floats), string manipulation, and streaming. As with Slopstone, we split the benchmark results to give a better overview.

In the case of medium-level operations we get an overall slowdown of around 12X, as shown in 6.3. This falls in line with the results of the previous benchmark if we take



Benchmark	Pharo JIT	Pharo NO-JIT	Bee
Overall	0.7	2.34	11.98
Strings and colls.	1	3.17	14
Primes	0.75	1.83	12.86
Fractonacci	0.73	4.17	12.5
Arithm. and colls.	0.67	2.97	6.41

Figure 6. Normalized Smopstone execution times , relative to host virtual machine (lower is better).

into account that the slowest results highly drag performance down.

7. Related work

Squeak[13] is a self-hosted Smalltalk implementation. The code of the virtual machine is written in *slang*, a subset of Smalltalk. Slang code is automatically translated to C source and then compiled with a C compiler, allowing for very good performance. Yet, the code written in slang is not object oriented and more difficult to understand and modify than standard Smalltalk code. Programmers have to be familiar with C programming, compiling and debugging tools.

PyPy[17] is another example of a self-hosted virtual machine. PyPy consists of an interpreter and a translation framework. The interpreter code is written in RPython, a restricted subset of Python. Unlike with slang, PyPy's translator operates on RPython source through many stages of analysis and optimization. Different backends allow generation different outputs. The main backend writes C sources.

Jalapeño/Jikes RVM [2, 3] is a research project that implements a Java virtual machine in Java. Jikes implements different types of garbage collectors, supports multithreading and has different compilers that provide for adaptive optimization and highly efficient code. Access to object headers is done through *Magic*, a set of methods that are not implemented in Java but assembly, and allow direct access to memory and processor control.

Maxine JVM [21] is another Java virtual machine done in Java. While it shares many ideas with Jikes, Maxine distinguishes itself by its inspector, which lets the developer visualize and debug all the state of the virtual machine.

Tachyon [7] is a self-hosted Javascript virtual machine. Tachyon does not use a bytecode representation, it compiles directly to machine code. The compiler operates on different intermediate representations, applying different optimizations. To augment the semantics of the language, Javascript syntax is extended with type annotations and primitives that allow direct access to memory.

Klein is a metacircular virtual machine for Self written in Self[6, 20]. It enjoys a fully object-oriented design. Through the use of mirrors it achieves great code reuse and is able to access meta-object properties. Thanks to this, Klein can be remotely debugged from other PCs. Reactivity is highly appreciated and the environment provides many tools to create the illusion that the system is made of tangible, physical stuff.

8. Conclusions

Bee project was started with the implementation of a JIT compiler that recreated the host virtual machine's one but that was written in Smalltalk. The success in doing so brought the question of what other parts could also be directly implemented within the language. Access to the JIT compiler allowed the usage of underprimitives, which leveraged the implementation of the rest of the system. Today, Bee is far from finished, yet we know all required functionalities can be implemented. Furthermore, the resulting code is fully object oriented and can take advantage of all the benefits that a high-level environment brings.

The main remaining question to be answered is what is the maximum performance to expect from the system. We believe that the answer to that question will be highly positive, and that we will be able to unravel the mystery very soon.

9. Future work

Being such a big project, many ideas are still left to be explored. Garbage collection is ready to be plugged to the system, but requires some modifications to allow running in the self-hosted bootstrapped system.

Debugging of the self-hosted system is also not possible. Browsers, inspectors and debuggers are available while in the hosted system but not in the bootstrapped one. To make them work we have to implement a messaging system that wraps the one brought by the hosted environment.

We also plan to support out-of-process debugging and inspecting. This will allow us to run on resource-limited systems via remote debugging, even in places where graphical environments are not be supported.

Current implementation of Bee is more than 10x slower than the hosted environment, while only implementing small optimizations. Work on polymorphic inline caches, and the optimizing compiler will provide a big boost in performance.

Bee has initial support for native multithreading. While we have not deeply explored the subject, we believe this will provide bigger performance improvements and also ease the implementation and exploration of non-blocking and asynchronous message sending.

References

[1] P. C. Allen Wirfs-Brock. A smalltalk virtual machine architectural model. Technical report, Instantiations, Inc., 1999.

[2] B. Alpern, C. R. Attanasio, J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. Shepherd, S. E. Smith, V. Sreedhar, H. Srinivasan, and J. Whaley. The jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000. ISSN 0018-8670. .

[3] B. Alpern, S. Augart, S. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, V. Sarkar, and M. Trapp. The jikes research virtual machine project: Building an open-source research community. *IBM Systems Journal*, 44(2):399–417, 2005. ISSN 0018-8670. .

[4] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. *Pharo by Example*. Square Bracket Associates, 2009. ISBN 978-3-9523341-4-0. URL `http://pharobyexample.org`.

[5] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for self, a dynamically-typed object-oriented programming language. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, PLDI '89, pages 146–160, New York, NY, USA, 1989. ACM. ISBN 0-89791-306-X. . URL `http://doi.acm.org/10.1145/73141.74831`.

[6] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of self a dynamically-typed object-oriented language based on prototypes. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '89, pages 49–70, New York, NY, USA, 1989. ACM. ISBN 0-89791-333-7. . URL `http://doi.acm.org/10.1145/74877.74884`.

[7] M. Chevalier-Boisvert, E. Lavoie, M. Feeley, and B. Dufour. Bootstrapping a self-hosted research virtual machine for javascript: An experience report. In *Proceedings of the 7th Symposium on Dynamic Languages*, DLS '11, pages 61–72, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0939-4. . URL `http://doi.acm.org/10.1145/2047849.2047858`.

[8] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '84, pages 297–302, New York, NY, USA, 1984. ACM. ISBN 0-89791-125-3. . URL `http://doi.acm.org/10.1145/800017.800542`.

[9] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.

[10] B. Hayes. Ephemerons: A new finalization mechanism. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '97, pages 176–183, New York, NY, USA, 1997. ACM. ISBN 0-89791-908-4. . URL http://doi.acm.org/10.1145/263698.263733.

[11] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 326–336, New York, NY, USA, 1994. ACM. ISBN 0-89791-662-X. . URL http://doi.acm.org/10.1145/178243.178478.

[12] U. Hölzle and D. Ungar. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Trans. Program. Lang. Syst.*, 18(4): 355–400, July 1996. ISSN 0164-0925. . URL http://doi.acm.org/10.1145/233561.233562.

[13] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: The story of squeak, a practical smalltalk written in itself. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '97, pages 318–326, New York, NY, USA, 1997. ACM. ISBN 0-89791-908-4. . URL http://doi.acm.org/10.1145/263698.263754.

[14] S. Kell and C. Irwin. Virtual machines should be invisible. In *Proceedings of the Compilation of the Co-located Workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, & VMIL'11*, SPLASH '11 Workshops, pages 289–296, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1183-0. . URL http://doi.acm.org/10.1145/2095050.2095099.

[15] G. Krasner. *Smalltalk-80 : bits of history, words of advice*. Addison-Wesley series in computer science. Reading, Mass. Addison-Wesley Pub. Co. cop.1983, 1983. ISBN 0-201-11669-3. URL http://opac.inria.fr/record=b1091689.

[16] F. Rastello. Ssa-based compiler design. 2015.

[17] A. Rigo and S. Pedroni. Pypy's approach to virtual machine construction. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 944–953, New York, NY, USA, 2006. ACM. ISBN 1-59593-491-X. . URL http://doi.acm.org/10.1145/1176617.1176753.

[18] L. Torczon and K. Cooper. *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2011. ISBN 012088478X.

[19] D. Ungar, R. Smith, C. Chambers, and U. Holzle. Object, message, and performance: how they coexist in self. *Computer*, 25(10):53–64, Oct 1992. ISSN 0018-9162. .

[20] D. Ungar, A. Spitz, and A. Ausch. Constructing a metacircular virtual machine in an exploratory programming environment. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 11–20, New York, NY, USA, 2005. ACM. ISBN 1-59593-193-7. .

[21] C. Wimmer, M. Haupt, M. L. Van De Vanter, M. Jordan, L. Daynès, and D. Simon. Maxine: An approachable virtual machine for, and in, java. *ACM Trans. Archit. Code Optim.*, 9(4):30:1–30:24, Jan. 2013. ISSN 1544-3566. . URL http://doi.acm.org/10.1145/2400682.2400689.

Embedding Multiform Time Constraints in Smalltalk

Ciprian Teodorov
Lab-STICC, ENSTA Bretagne, Brest, France
ciprian.teodorov@ensta-bretagne.fr

Abstract

Most of today's general-purpose programming languages include primitives for concurrent and parallel software development. However, they fail to provide mechanisms for reasoning about the complex interactions of the systems components. Amongst different formalisms, used for capturing the emerging and intricate characteristics of concurrent and parallel systems, the logical time model is widely used and proved useful in hardware, embedded and distributed systems domains.

In this study, we propose a meta-described clock-constraint engine, which embeds a formal model of logical time into the Smalltalk general-purpose language and environment. This engine, named ClockSystem, relies on the primitives provided by Clock Constraint Specification Language (CCSL) to provide a simple yet powerful toolkit for logical time specifications. ClockSystem extends the CCSL language, through an automata-based approach, with domain-specific user-defined operators and provides an embedded DSL for writing executable specification in a language close to the abstract CCSL notation.

The approach is symbiotic and benefits from the complementarity of the two languages. CCSL gains a readable syntax for library specification and the power of a highly dynamic general-purpose language and development environment. The Pharo Smalltalk environment acquires a very expressive time reasoning formalism, which promises improved software quality through formal verification and highly automated testing and monitoring strategies.

Categories and Subject Descriptors D.2.1 [*Software Engineering*]: Requirements/Specifications—Languages; D.2.4 [*Software Engineering*]: Software/Program Verification—Model checking

Keywords logical time, concurrency, domain-specific language

1. Introduction

In the context where multi-core heterogeneous computing became ubiquitous, and more and more support for concurrent and parallel applications is offered by todays programming languages. All application designers are faced with challenges that were once specific to the hardware, embedded or distributed domains. Amongst these challenges, there is the need to reason about time to create different levels of consensus between concurrent and parallel execution threads that have to communicate, to synchronise or to access shared external resources.

In the computer-science literature, different interpretations of time are studied, each one addressing particular aspects of the "real" time, which is known also as *physical time*. Amongst these interpretations, the logical clock model, introduced by L. Lamport [14], abstracts the notion of *physical time* to the partial order of events. This theory of *logical time* evolved since its beginnings, and today it is widely used in the hardware, embedded and distributed systems theory and practice.

General-purpose programming language, in their standard settings, offer a large variety of models, techniques and primitives to address concurrent and parallel programming. However, they fail to provide a mechanism for "time-aware" modeling, which proved very useful in these other disciplines of computer science and engineering.

Moreover, it is interesting to see that even the relatively young field of Model-Driven Engineering (MDE) recognised the importance of time and integrated it through the Clock Constraint Specification Language (CCSL) [1] in the UML Marte profile [20], which targets critical system modeling.

The CCSL formalism departs from the traditional approaches by offering a simple yet powerful logical time specification formalism, through a declarative domain-agnostic language. This formalism is integrated in the MDE infrastructure and tools through the TimeSquare toolkit [7] which provides a concrete-syntax for the CCSL language and a set of analysis tools targeting it, eg. simulator, model-animator. However, while the TimeSquare toolkit delivers powerful tools for system design and analysis it fails to offer a simple and readable syntax for specifying domain-specific libraries on top of CCSL. Moreover, the development efforts behind the TimeSquare toolkit are geared towards MDE users and besides offering the possibility to execute java code associated with clock events, it seems to lack an API for embedding CCSL specifications in Java, its implementation language [7].

In this study we introduce a *meta-described clock constraint engine*, named CLOCKSystem, which addresses these problems by providing an embedding of the logical time formalism in the Smalltalk object-oriented and general-purpose programming language and environment. Our system delivers an automata-based interpretation of the CCSL language formalism and allows extending the language constructs with user-specific primitives. Moreover, we have created an embedded domain-specific language (eDSL) for CCSL, which, through the use of the Smalltalk's flexible language, provides a simple, readable and extensible concrete-syntax for logical time specification. Furthermore, we argue that this eDSL is much closer to the abstract "pseudo-mathematical" notation presented in the CCSL literature than the one integrated in the TimeSquare toolkit.

One important goal behind our approach, besides embedding time in a programming language, is to offer the tools for opening the toolbox and enable to explore new ideas, identify new problems that could be addressed by such a toolkit, and provide a way to explore these new (or old for that matter) usage scenarios. To emphasise the advantages of our approach, some usage scenarios enabled by CLOCKSystem are presented. Amongst these, the possibility to exhaustively explore the state-space of a given specification paves the way to verification by model-checking [3]. The scope of this verification can either be the CCSL specification alone or its composition with the time-constrained-system, which however has to be expressed in a formal language. A second usage scenario, namely Design-Space Exploration, is enabled by the complementarity between the declarative CCSL formalism, enabling the simple encoding of specifications, and the imperative nature of Smalltalk, which offers the power of a general-purpose programming language for "scripting" different analyses phases searching throughout the solution-space. Testing and Monitoring are other usage scenarios in which the CLOCKSystem specifications are simply seen as the compact encoding of a test case which can, also, be deployed in production and which follows the system execution (through events) constantly checking the coherence between the specification and the real execution observed.

To better illustrate our approach, throughout this study we use an example based on a logical clock specification of a simple Synchronous Data-Flow model of computation, which is inspired from [18].

The main contributions of this study are:

- The integration of a logical clocks formalism into a general-purpose object-oriented programming language like Smalltalk;
- The design of an small and extensible logical time kernel, which, while based on the CCSL language, extends its expressiveness though the addition of automata-based user-defined primitives;
- The creation of a readable and extensible embedded DSL for the creation of domain-specific parametric libraries of clock relations. This eDSL uses simple Smalltalk message-sends for creating a skinnable language on top of a simple core interchange format;
- The introduction of a several scenarios for which ClockSystem has the potential to facilitate formal methods integration, and ultimately the creation of better software applications.

This study is structured as follows. Sec. reviews the main motivations behind our approach while introducing some of the related work. The CLOCKSystem eDSL is introduced in Sec. 3 and it is compared with the abstract CCSL and the TimeSquare notations. Sec. 4 presents the intuition behind four different usage scenarios enabled by our approach. Sec. 5 discusses some details of our toolkit, presenting its core structure, the extension mechanism, an interchange format and the details of our eDSL concrete-syntax before briefly introducing the semantics. This study concludes in Sec. 6 overviewing some future research directions.

2. Motivations and Related Work

This sections overview the main motivations and principles that have driven our approach. First the notion of time and some of its interpretations are briefly overviewed. Then the Clock Constraint Specification Language is introduced along with the TimeSquare toolkit, the de-facto CCSL implementation. Some of the limitations of the current tools are presented, emphasising the need for an user-friendly concrete-syntax and a more natural embedding in a general-purpose programming language. Finally, some technical advantages emerging from the complementarity between a declarative language, such as CCSL, and an imperative high-level language, such as Smalltalk, are presented.

2.1 From Time to Logical Time

To cope with the complexity of the intricate relations between time and other concepts that they manipulate, different disciplines often use particular interpretations of time. Schreiber addresses some of the fundamental issues of the notion of time, in the context of computer science and engineering, in [24] .

A very important distinction that govern much of our current view of time is the distinction between the quantitative notion of time in physics, sometimes referred as "physical time" in computer science literature, and other more abstract models capturing only some characteristics of the physical time and its influences (ex. relations). For example, in today's digital integrated circuits, time is approximated using the discrete notion of clock. A clock is a particular type of circuit that oscillates periodically between two distinct values used to coordinate the operation of digital circuits. To cope with their complexity, the designers divide the circuits in different clock-domains each one driven by an

independent clock, hence creating multi-clock systems, often called polychronous [11]. The communications between these clock-domains are based either on clock synchronisation or on handshake protocols. Both these techniques are equally found in concurrent and distributed systems and pose unique challenges for reasoning about the system actions. In these two particular cases, what counts is not so much the time itself (its physical representation, nor its discrete interpretation with clocks) but the events of interest (sending/receiving a value, waiting for a partner, etc) and their partial ordering. To capture these aspects, in his seminal work [14], L. Lamport introduced the notion of logical clocks which abstracts away the notion of physical time to a partial order of events of interests. The connection between these logical clocks and the causal relations between the corresponding events identified by R. Schwarz et al. [25] gave rise to a rich theory enabling to characterise the behavior of distributed systems. Moreover, since the nature of the events of interests is not necessarily time related, this theory enables reasoning about other physical quantities and concepts. A typical example is the notion of deadline which can be expressed either as the process should stop after 15 sec. or the process should stop when reaching 80 degrees celsius. In the latter case, we use a logical clock to follow the evolution of temperature and stop the process when the deadline is met. This generalisation of logical clocks to other physical or abstract quantities (modelled as events) is known also as Multiform Time [21].

Today, highly-complex multi-core computing architectures are ubiquitous. They enable the concurrent and/or parallel execution of thousands (if not millions) of software tasks (be them processes, threads, actors, etc.). In this context, the need for time-driven reasoning permeates more and more from the hardware and distributed system domains to mainstream software development. Take for example the highly complex interactions between the execution threads of a typical web-browser. In these cases relying on logical clocks as a model of time has the potential to greatly improve software quality by enabling formal reasoning and verification. However, while all of todays general-purpose programming languages include primitives for concurrency and parallelism through different mechanisms, in standard settings, none of them offers support for time-aware modelling, reasoning or verification that proved very useful in the context of hardware, distributed and realtime system modelling and implementation. The CLOCKSystem language and toolkit tries to address this shortcoming by embedding a logical time formalism, namely CCSL, into the Smalltalk general-purpose object-oriented language and environment.

2.2 Clock Constraint Specification Language

The notion of Logical time is at the core of synchronous languages, such as Signal [15] and Lustre [12], and they are extensively used for the design and implementation of hardware and embedded real-time systems. However, directly integrating such approaches into a general-purpose programming language posses many challenges, mainly due to the complexity of these languages and the presence of technical artefacts coming from the embedded domain. The CCSL language [1], was designed to represent time relations through the logical time formalism following a high-level domain-agnostic approach. Hence, it makes an ideal target for embedding, since it is conceptually simple, and free of technical-space artefacts.

The core abstraction of CCSL reposes on the notion of $clock$, viewed as a strictly ordered sequence of instants (ticks), and the explicit descriptions of the relations between the instants of a set of $clocks$. There are two principal classes of relations: causal and temporal. The basic causal relation is the $precedence$ relation ($a \preceq b$) implying that the instants of the clock a causes the instants of clock b. The main temporal relations are the: $coincidence$ ($a = b$), meaning that both the instants of a and b occur at the same time or do not occur at all; $strict\ precedence$ ($a \prec b$), meaning that the instants of a always occur before the ones of b and never at the same time; and the $exclusion$ ($a \# b$), stating that the instants of a are mutually exclusive with the ones of b. Besides these core relations, CCSL defines a $subclocking$ relation ($a \subset b$) used for specifying that the set of instants of clock a is actually a subset of the instances of clock b – whenever an instant of a occurs, an instant of b occurs.

To enable the characterization of complex clock specifications, the CCSL language introduces a number of clock expressions that, as opposed to the relations, enable to derive new clocks based on the existing ones. Some of these expressions are: the $intersection$ ($a * b$), which creates a clock having the set of instant equal to the intersection of the set of instants of the arguments; the $union$ ($a + b$), which derives a new clock based on the union of the set of instances of the arguments; the $infimum$ ($a \wedge b$) and $supremum$ ($a \vee b$) which defines a new clock faster/slower than both arguments (coincident with the fastest/slowest); the $waiting$ ($a\ \$\ n$), creating a clock that ticks only after n ticks of the argument clock.

Another interesting, and rather complex expression is the clock $filtering$ ($a \blacktriangledown o.(p)^{\omega}$) that creates a new clock coincident with explicitly selected instants of the argument clock a. These instants are selected based on a specification encoded as a binary word ($o.(p)^{\omega}$) composed of two distinct binary sequences: the $offset$ (o), seen as a static non recurring sequence, and the $period$ (p) a infinitely repetitive binary word. The instants of this clock follows closely the structure of these two binary words. For each instant of the argument clock a we move to the right in the sequence, if the bit is set to 1 the resulting clock should tick if not it should not. Once at the end of the periodic sequence we restart from the beginning of this sequence.

The TimeSquare toolkit [7] is the de facto standard toolkit for the specification and the analysis of CCSL logical time

specifications. It is implemented as a model-based environment integrated into the Eclipse platform, and benefits from a number of model-driven technologies. TimeSquare proposes a concrete, textual syntax for the CCSL language based on XText DSL framework [9]. Besides, TimeSquare implements a CCSL constraint resolution engine for simulating the specifications, integrates model-animation facilities and offers the possibility to execute arbitrary Java code symbolically associated to clocks from the specification.

The CCSL language provides a very expressive formalism for reasoning about logical time and the intricate relations between events in real systems. Moreover, TimeSquare enables the definitions of domain-specific libraries build from the primitive operators. However, in some cases, the declarative and sometimes complex nature of the CCSL primitive operators renders the creation of these libraries difficult, and even inefficient with respect to the complex constraint resolution policy needed for implementing its semantics. To address these issues, in CLOCKSystem we introduce the possibility to extend this core language, through domain-specific user-defined automata. A side-effect of this capability is the possibility to explicitly meta-describe all CCSL primitive operators and include them simply as a standard library, instead of hard-coding their exact semantics in the execution engine.

Furthermore, the XText-based concrete-syntax integrated in the TimeSquare toolkit, while having its advantages, renders the task of library specification difficult due mainly to an important syntactic overhead compared to the abstract notation presented in the literature. CLOCKSystem addresses this issue by providing an extensible, simple eDSL implemented through Smalltalk messages which through the use of syntactic synonyms can be adapted to domain-specific vocabularies or even user preferences. Moreover, it proposes a simple interchange format as a common basis for bridging the gap between possible vocabulary differences and for interoperability with external environments.

2.3 Opening the toolbox

To achieve our goal of integrating logical time in a general-purpose programming language, we need to open the toolbox and expose the core of the formalism along with the associate tooling to the host environment. Through the eDSL proposed by ClockSystem, which uses syntactically correct Smalltalk code for CCSL specifications, we move one step closer towards this goal. However, the real gain comes from the new usage scenarios that emerge due to the possibility to run arbitrary pre-preprocessing and post-processing steps on any given specification, to link logical time-models with a dynamic environments such as Smalltalk and to provide the application developers with tools for reasoning about intricate concurrency problems. We believe, that an approach such as CLOCKSystem can serve as a basis for studying and understanding better the relations between our programming environments and the highly complex systems on which they run on. At the same time, CLOCKSystem is an experimental platform for improving the quality of current models of time which have a number of shortcomings, such as: *a)* poor scalability for large models; *b)* poor support for dynamic systems.

3. CLOCKSystem for CCSL Users

An important requirement for implementing a modelling language as an embedded DSL (eDSL) in a general-purpose programming language is that the embedding should reduce the syntactic overhead to a minimum. Hence, providing a comfortable and familiar environment for the DSL users, while at the same time enabling the eDSL designers to focus more on the language features than on the grammar development and parsing. An embedding is not always perfect, and often some amount of syntactic overhead is inherent. To emphasise our results we compare our syntactic encoding of the CCSL model with the abstract notation, introduced in different papers, and the TimeSquare language. Towards the end of this section, we show that in the cases where our encoding fails to match the abstract-notation it reuses the textual encoding of TimeSquare. Moreover, our lightweight syntax, based on message sends, enables the user to easily define keyword synonyms that can help to close the gap between a given domain-specific vocabulary and our formalism. We illustrate the results of our embedding of CCSL in Smalltalk (Pharo dialect) through a simple example inspired from [18]. This example is focused on the modeling the control aspects of Synchronous Data-Flow (SDF) applications with CCSL.

3.1 Case Study: Synchronous Data-Flow

SDF graphs are an abstraction for modeling data-flow computations that enables static task scheduling. This model encodes data-flow computation as a graph where nodes represent the computations (actors) and the edges represent the data dependencies. The designer associates to each computation block the static rates of input consumption and output production for each input/output dependencies. A simple SDF model can thus be represented with a graph with the edges labelled with 3-tuple (outputRate, initialTokens, inputRate). Note that here the storage capacity of each edge is infinite, as in the case of Kahn networks [13].

The execution of a SDF application is governed by the following rules:

- An actor can execute (is enabled) only when all its required inputs are *available*. An input is *available* when the number of tokens (data samples) in the incoming edge is larger or at least equal to the predefined inputRate;
- The execution of an actor results in the consumption of $inputRate$ tokens from all incoming edges and the production of exactly the $outputRate$ tokens on each of its output edges. The tokens produced by one execution are buffered on the outgoing arcs in a First-In First-Out (FIFO) manner;

- $initialTokens$ is a statically defined property of edges defining the number of tokens available at the beginning of the execution;
- The execution of any actor is not dependent on the token values, meaning that the control is data-independent.

In [18] the authors describe one possible CCSL encoding of these execution rules using three clock constraints describing the allowed actor firings. This encoding associates to each actor a CCSL $clock$ representing the execution of the actor. The FIFO channel (edge) between two actors are managed with another two $clocks$: $read$ and $write$. The $read$/$write$ clock ticks whenever one input/output is added/removed to the FIFO. Then for each channel three constraints on these clocks are added: *1)* $input$ constraint, governing the relation between the actor execution and the $inputRate$ tokens available at the input; *2)* $output$ constraint, governing the relation between the actor execution and the $outputRate$ tokens produced; *3)* $token$ constraint, encoding the number of available tokens in an arc as the difference between the number of read and write operations.

3.2 Constraint Definition Syntax: Comparative Study

The CCSL encoding of the $input$ constraint is specified in [18] as a precedence relation using one $precedence$ relation and one $filteredBy$ expression. Listing 1 shows the encoding of this constraint using the abstract notation. The intuition behind this constraint is that the actor execution should be preceded by the addition of at least $inputRate$ tokens in the channel.

Listing 1: CCSL specification for the SDF input constraint

```
1 def input(clock actor, clock read, int inputRate) ≜
      (read ▼.(0^(inputRate−1).1)^ω) ≺ actor
```

In CLOCKSystem the input constraint (from Listing 1) is expressed by defining a message `input:read:inputRate:` implemented like in Listing 2, where $actor$ and $read$ are clocks and $inputRate$ is a number. The message `period:` can be seen as syntactic sugar defined to create a $filterBy$ expression without an $offset$. The binary word required by the expression is created by using classical Smalltalk Array concatenation (the `for:` message send to a number X creates an array with n identical elements equal to X). The `<` message represents exactly the precedence relation as the $\prec$ abstract notation.

Listing 2: CLOCKSystem specification of the SDF input constraint

```
input: actor read: read inputRate: inputRate
 (read period: (0 for: (inputRate-1)),{1}) < actor
```

The reader should notice that the principal reason for the syntactic overhead in Listing 2 comes from the representation of special characters and notations, such as ▼, and power notation x^y as ASCII encoded message sends (`period:, for:`). Besides that, there are two Smalltalk-specific artefacts, namely the $colon$ separating parts of the message symbol, and the $comma$ that replaces the dot character in the abstract notation. These represent a small syntactic overhead that will probably not be present in a CCSL-specific keyword-based language grammar. Notice also that the `0 for: (inputRate-1)` does not use the common ^ symbol used for power notation in some general purpose programming languages since it is a Smalltalk reserved character. Nevertheless, we consider that in this case our notation follows rather closely the abstract one, especially when compared to the rather verbose language used in TimeSquare for the same purposes, see Listing 3. We will leave to the reader the exercise of understanding the meaning of that Listing.

Listing 3: TimeSquare specification of the SDF input constraint

```
RelationDeclaration Input(
        actor:clock,
        read:clock,
        inputRate:int)
RelationDefinition InputDef[Input]{
        Sequence ByInputRate=
                (IntegerVariableRef[inputRate])
        Expression readByInputRate=FilterBy(
                FilterByClock->read,
                FilterBySeq->ByInputRate)
        Relation inputRateTokenExec[Causes](
                LeftClock->readByInputRate,
                RightClock->actor)
}
```

Listing 4 shows the composition of the CCSL relations needed for representing the SDF semantics. We will not describe the meaning of this listing since it is very well explained in [18]. However, for comparison we show the CLOCKSystem equivalent in Listing 5, and note the small syntactic overhead, again compared to the TimeSquare specification which amounts for almost 100 lines of code and was not included for obvious reasons.

Listing 4: CCSL specification of the SDF semantics

```
1 def edge(clock source, clock target,
        int out, int initialTokens, int in) ≜
        clock read
        clock write
        source = (write ▼.(1.0^(out−1))^ω)
6       ∧ write ≺ read $ initialTokens
        ∧ (read ▼.(0^(in−1).1)^ω) ≺ target
```

Listing 5: CLOCKSystem specification of the SDF semantics

```
edgeFrom: source to: target
    outRate: out initial: initialTokens inRate: in
3   |r w|
    r := self localClock: #read.
    w := self localClock: #write.

    source===(w period: ({1}, (0 for: (out-1)))).
8   w  < (r waitFor: initialTokens).
    (r period:  (0 for: (in-1)), {1})  < target
```

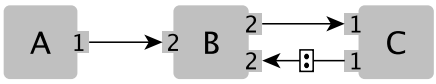


Figure 1: An example of an SDF graph

3.3 Constraint Instantiation

In the last section we have presented the creation of a library operator for encoding SDF execution as CCSL clock and clock constraints. In this section, we illustrate the usage such an operator in the case of the simple SDF application in Fig. 1. This example consists of three actors A, B, and C connected with three edges labelled as follows: $E_{AB}(1,0,2)$, $E_{BC}(2,0,1)$, $E_{CB}(1,2,2)$.

Using the CCSL abstract notation it suffices to instantiate the *edge* constraint as follows: $edge(a,b,1,0,2) \wedge edge(b,c,2,0,1) \wedge edge(c,b,1,2,2)$. In CLOCKSystem, the same effect can be achieved through a script like the one in Listing 6. For brevity, we omit the clock definitions in the CCSL case (one for each SDF actor: a, b, c). The CLOCKSystem notation is more verbose compared to the abstract one, which is due to the use of multi-arguments message sends. In our case, this overhead is not strictly necessary, and can be seen as a personal choice, but we believe that it improves the readability of our specifications. The alternative would be to use an array encoding of the arguments (such as $edge: \{a.b.1.0.2\}$) which would be much closer to the CCSL notation.

Listing 6: CLOCKSystem instantiation of the SDF constraints for the example in Fig. 1

```
sys := ClockSystem named: 'sdf'.
a := sys clock: #A.
b := sys clock: #B.
c := sys clock: #C.

sys
  edgeFrom: a to: b outRate:1 initial:0 inRate:2;
  edgeFrom: b to: c outRate:2 initial:0 inRate:1;
  edgeFrom: c to: b outRate:1 initial:2 inRate:2.
```

In TimeSquare, the instantiation is done in a similar way, however with some complications brought by the integration with the UML Marte profile (ex. the clocks have references to the model elements). In [6] the authors presents an extension of the OCL language, named ECL, enabling the creation of CCSL instantiation scripts that would then be executed on particular model instances. For this aspect the similarity between OCL constructs with the traditional Smalltalk API (especially the Collection API) makes us conclude that the user of CLOCKSystem has at his disposal a much richer "scripting" language which can be used for the same purposes as ECL.

Table 1: Syntactic differences between CCSL notation, CLOCKSystem and TimeSquare.

Name	Notation	CLOCKSystem	TimeSquare
Subclocking	$a \subset b$	a subClock: b	SubClock(a, b)
Coincidence	$a = b$	a === b	Coincides(a, b)
Precedence	$a \preceq b$	a <= b	NonStrictPrecedes(a, b)
Strict Precedence	$a \prec b$	$a < b$	Precedes(a, b)
Exclusion	$a \# b$	a <> b	Exclusion(a, b)
Expressions			
Inf	$a \wedge b$	a inf: b	Inf(a, b)
Sup	$a \vee b$	a sup: b	Sub(a, b)
Defer	$a\ (ns) \rightsquigarrow b$	a defer: b for: ns	Defer(a, b, ns)
Sampling	$a \mapsto b$	a nonStrictSample: b	NonStrictSample(a, b)
Strict Sampling	$a \rightarrow b$	a sample: b	Sample(a, b)
Intersection	$a * b$	$a * b$	Intersection(a, b)
Union	$a + b$	$a + b$	Union(a, b)
Waiting	$a \$ n$	a waitFor: n	WaitFor(a, n)
Preemption	$a \lightning b$	a upTo: b	UpTo(a, b)
Filtering	$a \blacktriangledown o.(p)^{\omega}$	a filterBy: {o.p}	FilterBy(a, b)

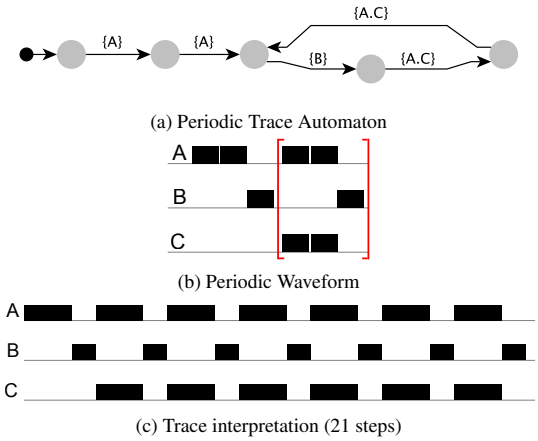
3.4 Syntactic Differences and Synonyms

To complete our comparison, Table 1 shows some of the most important operators of the CCSL language using the abstract, CLOCKSystem and TimeSquare notations. In CLOCKSystem, the strict precedence, intersection and union relation use the same notation as the CCSL description. The precedence uses the widely accepted ASCII encoding for $\preceq$. For the coincidence and the exclusion relations different notations were used due to the use of = for equality checks in Smalltalk language, and the reserved use of the # character. In these cases we also defined synonym messages that reproduce the TimeSquare naming. All other CCSL operators are encoded using a camel-case version of the TimeSquare keywords. A particular case is the `defer:for:` message, which uses a multi-argument message for the same readability reasons we explained in the case of `edgeFrom:to:outRate:initial:inRate:` (Listing 6).

The CLOCKSystem encoding of all CCSL operators as message-sends enables the user to easily define keyword synonyms by simply defining a new message that redirects its arguments as needed to the provided primitives, see for example Listing 7 showing 4 equivalent ways of creating a strict precedence relation between two clocks a and b. This feature is clearly a by-product of our embedding, however it is very important for a modeling language as generic as CCSL since it enables the users to adapt the specification language to match the vocabulary of their domains of interest or their personal choices.

Listing 7: Syntactic synonyms for $a \prec b$ relation

```
a < b.
b > a.
a precedes: b.
system relation: #strictPrecedence clocks: {a. b}
```



(a) Periodic Trace Automaton

(b) Periodic Waveform

(c) Trace interpretation (21 steps)

Figure 2: Cyclic simulation trace and different visualisations with CLOCKSystem for the SDF example in Fig 1

4. Beyond Standard Simulation

While different use-cases for CCSL were proposed in the literature [17, 26], currently the main functionality implemented in TimeSquare is the simulation of specifications, with the possibility to animate different model elements by associating clock ticks with the execution of particular functions. In this section, we overview some extensions and new usages that are enabled by our embedding in the Smalltalk environment.

Cyclic Trace Interpretation. The CLOCKSystem simulator implements a trace-based simulator. While executing a given specification, it constantly verifies the existence of loops back to an already seen system state, in which case it can either stop the simulation reporting an infinite trace (infinite due to the possibility to loop-back an arbitrary number of times) or it can continue, maybe choosing a different path. Fig 2 presents the results obtained for the SDF example, introduced in Fig. 1. The first visual representation of the executions trace, in Fig. 2a, offers an automaton view of the simulation trace, while the second one in Fig. 2b) shows a different waveform-like visualisation which uses the square brackets to represent the unbounded repetition of the last 3 steps. Traditionally, the TimeSquare simulator is producing a waveform trace similar to the one we present in Fig. 2c. However, in our case this finite simulation trace was obtained by the interpretation of the automaton presented in Fig. 2a for exactly 21 steps, and not directly from the CCSL specification.

Exhaustive Reachability Analysis and Model-Checking. Besides the simulator, the CLOCKSystem toolkit provides the possibility to perform exhaustive reachability analysis of the CCSL specifications thus paving the way towards formal verification of properties agains these specifications.

To better understand the importance of providing such facilities, consider for example the approaches taken in [26] and [19] for model-checking UML Marte application restricted by CCSL constraints. In these two cases, the authors invested a lot of effort to encode (more or less manually) the correct semantics of each CCSL operator in a formal language, such as Fiacre [10], moreover the complex constraint composition mechanism had to be implemented in those languages. We believe that this process is cumbersome, and prone to errors especially since these two formalism are more adapted for asynchronous system modeling and verification. As such, another degree of difficulty was added by the interpretation of the coincident clock firings as the interleaving of all events. Moreover the property specification, and the result interpretation in these cases is difficult since the resulting semantic encoding was polluted by the semantics of the constraints and constraint composition encoding.

Relying on the exhaustive reachability results, we have developed an interface with the OBP model-checking toolkit [8] that enables the verification of UML models. To achieve this, an UML model is transformed to a formal language (as in the previous cases) and the resulting program is composed with the reachability analysis results produced by CLOCKSystem. To ensure the correct semantics for the composition, the results obtained with CLOCKSystem were post-processed only for expanding the coincident relations (by generating the correct interleaving)[1]. This approach enables the verification of safety and bounded liveness property on a subset of UML Marte constrained using CLOCKSystem specifications.

Design-space Exploration. An important aspect during system design is creating a feedback-loop between a given system model and the analysis results. Conceptually simple, this process, known also as design-space exploration, states that the analysis results should be taken into account to improve the model. The automation of this process is hindered, in the case of declarative languages, by the lack of an adapted programming layer around the modeling language and associated tools (solvers, simulators, etc.), which drives the designers towards the use of complex and low-level script-based solutions, which are hard to create and maintain. Embedded DSLs rely on host-language facilities for the automation of such task, and, in the case of CLOCKSystem , the full power of the Smalltalk language and environment is at user disposal.

Testing and Monitoring. In a concurrent software context, the clocks could be seen as types of events which are produced during execution, then a CLOCKSystem specification describes the set of valid relations between these events. In

[1] We call coincident firings (relations) all cases where two clocks tick at the same time. Visually these cases are represented by tuples like `{A. C}` in Fig. 2a

this case, a program can be viewed as a high-level test specification, which encodes not only one valid execution path but a set of paths. Integrating such approach into unit testing frameworks such as SUnit does not pose any challenges, however it can help detect subtle concurrency bugs in concurrent Smalltalk applications. In production, these specifications could be embedded into the deployed images to help monitoring the application. Moreover, a counterexample, resembling the traces in Fig. 2, can be generated to help understanding the cause of the malfunction.

5. The CLOCKSystem Toolkit

The CLOCKSystem language is an extension of the CCSL domain-specific language (DSL). The implementation is deeply embedded in Pharo Smalltalk environment. As an embedded DSL, CLOCKSystem programs are encoded as syntactically correct Smalltalk code, moreover its abstract-syntax tree (AST) is exposed as plain smalltalk objects. While benefiting from the CCSL simple but powerful approach for time reasoning, CLOCKSystem exploits the flexibility of the Smalltalk language to provide a readable syntax for the CCSL language, that can replace the current library specification language integrated in TimeSquare.

The key ideas behind our approach are: *a)* provide a minimal kernel for experimenting with logical time formalisms in Smalltalk; *b)* offer a flexible and simple language for extending the kernel with user-defined event relations; *c)* enable the development of new analysis tools for CLOCKSystem specification, like exhaustive reachability analysis.

This section starts by describing the kernel of our environment, emphasising the possibility to extend the language primitives introduced by CCSL with user-defined clock-relations. Then a minimal core-syntax in presented, which can be used interchange format between different environments, before briefly discussing the execution semantics and some of the existing analysis tools.

5.1 Meta-described Clock Constraints

The CCSL language, was designed to represent time relations through the logical time formalism following a high-level domain-agnostic approach. Hence, since it is conceptually simple, and free of technical-space artefacts, it is an ideal candidate for introducing notions of time into a general-purpose programming language.

The Need For New Primitives. Nevertheless, we have found that relying only on the primitive operators provided by CCSL was sometimes inefficient, cumbersome and rendered the expression of state-based relations difficult.

To illustrate these difficulties, consider the SDF example introduced in the Sec. 3.1. In this case, an equally valid SDF execution semantics (as in Listing 4) can also be encoded using an automaton like the one presented in Fig. 3, in which case the $output$, $token$, and $input$ constraints (used in Listing 4) are encoded in a simple controller automaton governing the access to the FIFO channels connecting the actors. The intuition behind this automaton is as follows: *a)* reading and writing to the channel are exclusive – no reading and writing at the same time; *b)* the process writing data (represented by the w clock) simply writes $outRate$ tokens to the channel; *c)* the process reading data (represented by the r clock) is enabled only if there are enough tokens in the channel $size >= inRate$, otherwise it is blocked. In this case a specification for the SDF application in Fig. 1 only needs to create only 3 clocks and to instantiate 3 relations, one for each edge in the SDF application, instead of 18 clocks and 18 relations needed in the case of the specification presented in Listing 6. This renders the specification easier to understand, and speeds up the model simulation and analysis since it does not introduce intermediate clocks nor relations.

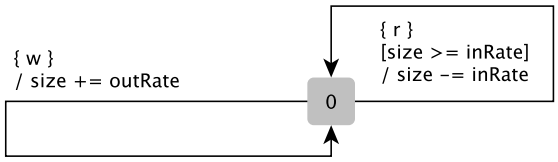


Figure 3: Automaton encoding the SDF execution policy

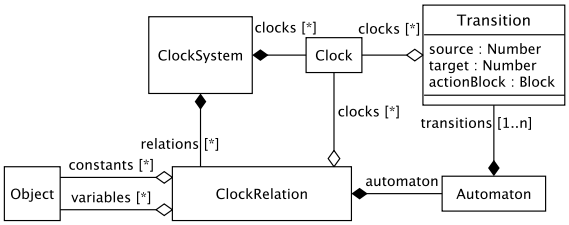


Figure 4: CLOCKSystem model (abstract syntax)

To address this expressivity problem, in CLOCKSystem we have decided to implement the CCSL operational semantics by specifying its mapping to a state-machine based encoding, such the one presented in [23], rather than directly implementing it in a traditional interpreter (as is the case in TimeSquare). This approach proved very useful since it enabled from the beginning the possibility of using automata-theoretic analysis techniques, such as reachability analysis and model-checking, directly on our model without recurring to complex model-transformation approaches (such the ones presented in [26]). Moreover, it helped reducing the number of language concepts to a minimum (all primitives operators are meta-described by automata), and opened the conceptual framework for seamlessly integrating state-based relations into the CLOCKSystem language.

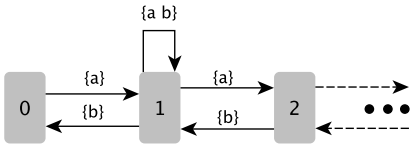


Figure 5: The infinite automaton of the $a \prec b$ relation

ClockSystem Metamodel. At its core, the CLOCKSystem toolkit relies on the Smalltalk implementation of the meta-model presented in Fig. 4. In this meta-model, the two central concepts are the $Clocks$ and the $ClockRelations$. The $Clocks$ are instantiated and linked to problem-space objects representing the different events of interests. Each $ClockRelations$ contains an automaton specification encoding its operational semantics. Conceptually, this automaton is just a set of transitions between discrete states. Each transition is just an association, between one source state and one target state, labelled by a vector of $Clocks$ that tick when the transition is executed and an actionBlock that is executed when the transition is fired. The purpose of this action block is to update either the state-variables of the automaton or the global variables in the system. Semantically, the execution of each transition is considered atomic. Note that, in our setting, the CCSL expressions are nothing more than simple $ClockRelation$ instances with an "internal clock" representing the clock produced by the expression. The $ClockSystem$ class, in Fig. 4 simply composes the set of $Clocks$ and $ClockRelations$ defined in a given model.

A DSL for Primitive Relations. Traditionally, in automata-based approaches for ensuring theoretical properties (such as decidability, termination, etc.) the state-machine are constrained to be finite. However, this is not the case in CCSL, which has some infinite clock relations, such as the precedence. To cope with this difficulty, in CLOCKSystem, infinite automata are encoded symbolically through a relation-definition DSL (relDSL) using Smalltalk blocks[2]. In this case, the $Automaton$ of given relation does not explicitly contain a set of transitions but a block that returning the outgoing transitions from a given state.

To better illustrate this aspect consider, for example, the infinite automaton for the strict-precedence relation shown in Fig. 5[3]. In CLOCKSystem this relation is defined by the Smalltalk block presented in Listing 8. The infinite number of states in the automaton is encoded through the state-variable s which is a Smalltalk integer. Once this encoding is in place, the block responsibility is to return the possible transitions from a given state. For example, if the state variable is 0, executing the block such as `strictPrecedence value: 0 value: clock1 value: clock2` will return a set with only one transition, namely {`s->(s+1) when: {a}`} saying that the automaton can go to the state s+1 (0+1 in this case) and if it does the $clock1$ should tick and $clock2$ should not. Note that in this case another transition is possible, namely `s->s when:` $\{\neg clock1.\neg clock2\}$ stating that the system can stay in the same state s for an indefinite period of time. However, if it does so, neither $clock1$ nor $clock2$ can tick. The CLOCKSystem execution engine automatically adds the negation off all clocks not present in a transition vector, and the transitions that block all clocks while staying in the same state of the system to ensure the correct semantics.

Note that, due to the unbounded representation of integers in Smalltalk, (through SmallInteger, BigInteger instances) limited only by the amount of available memory, we did not need to use a symbolic integer encoding, which might be more adapted in certain situation.

A side-product of this simple block-based representation is support for manipulating variables in the automatons that comes at no cost. The variables are nothing more than state-variables (such as s). Instead of interpreting them as the source/target of transitions they are used for building predicates to guard the transitions, and are updated in the $actionBlocks$ using plain Smalltalk code. Constants are also supported in the same manner. For constants, to ensure that they are not updated in the action-blocks they are simply not passed as arguments when these blocks are evaluated. They can, however, be used in a read-only manner since they will be free variables in the action block and capture their value from the enclosing scope, the automaton block – where they are block arguments which are not assignable in Smalltalk.

Listing 8: The CLOCKSystem definition of the infinite $a \prec b$ relation

```
1 KernelLibrary>>#strictPrecedence
    ^ [ :s :a :b |
    "unbounded strict precedence"
    s = 0
        ifTrue: [ {
6           s -> (s + 1) when: {a} } ]
        ifFalse: [ {
            s -> s when: {a. b}.
            s -> (s + 1) when: {a}.
            s -> (s - 1) when: {b} } ] ]
```

A Primitive for SDF. To illustrate the generality of our approach, consider once more the SDF example introduced in Sec. 3.1 and the possible automata-based relation specification introduced in Fig. 3. To encode this relation in CLOCKSystem, firstly we add a block argument s representing the mapping of the discrete automaton states to in-

[2] Note that relDSL can be seen as a meta-level DSL for specifying ClockSystem primitive relations and should not be confused with the ClockSystem DSL which only instantiate these relations

[3] All CLOCKSystem automatons are synchronous, and complete in terms of the clock vocabulary. To simplify the presentation we do not include in Fig. 5 the transitions that loop in a state while not enabling any clock nor the negation of all clocks not enabled by the transition

tegers. Then, the variables manipulated by the automaton are identified and added as arguments – $size$ in our case, followed by the constants used in the predicates – $inRate$, $outRate$ and $capacity$ (when the $capacity > 0$ we consider the the FIFO channel is bounded and can contain maximum $capacity$ token, obviously for a valid model the $capacity \geq outRate$). Lastly, we add the clocks that are constrained by the automaton – r and w in our case. Once the arguments of the block identified, the transitions are encoded in the block, see Listing 9 for this example. First of all, there is a slight difference from Fig. 3, introduced by adding the notion of channel $capacity$. The clock w and the associated transition is enabled only in the case where either the $capacity \leq 0$ – the channel is unbounded – or, if it is bounded, there is enough place in the FIFO to store $outRate$ tokens ($capacity - size >= outRate$). Note also the presence of the $actionBlocks$ used to update the variable $size$. As stated before, these blocks are executed when the corresponding transition is fired with the state-variables as arguments (in this case for example, the actionBlock is executed through as message send like `actionBlock value: currentState value: currentSize`, where s and size are the values of the state variables at a given point during execution).

Listing 9: User declared relation for a SDF channel

```
SDF >> #channel
    ^[:s :size :inRate :outRate :capacity :w :r |
    |transitions|
    transitions := OrderedCollection new.
    size >= inRate ifTrue: [
        transitions add: (
            0->0 when: { r } do:
                [:conf | |sz|
                    sz := conf at: 2. //size var
                    conf at: 2 put: (sz - inRate)
                ]) ].
    (capacity <= 0 or:
    [capacity - size >= outRate]) ifTrue: [
        transitions add: (
            (0->0) when: { w } do:
                [:conf | |sz|
                    sz := conf at: 2. //size var
                    conf at: 2 put: (sz + outRate)
                ]) ].
    transitions asArray]
```

All Relations are Not Created Equal. Using this encoding scheme we have been able to model all CCSL operators, except the concatenation operator. In automata-theoretic approaches the CCSL concatenation relation is known as the sequential composition of state-machines. Hence, even though in CCSL it is presented on equal terms with respect to the other clock relations, it is really a meta-operator that enables to link several clock relations in a sequential manner. In CLOCKSystemthe CCSL concatenation can be implemented by the explicit identification and annotation of the final states of the several finite relations. Then the concatenation relation instance is responsible only for passing the control from these final states to the initial state of the following automaton.

Some Practical Limitations. Though simple, and powerful, this technique has the disadvantage of rendering the state-machines opaque, making it difficult to statically reason about the primitive relations in CLOCKSystem. For example, it is hard extract the set of transitions of a given finite automaton. In the case of TimeSquare, and traditional CCSL this is not an issue due to the fix number of primitive relations, which can be hard-coded in an analysis engine. However, in our case such "hard-coding" is not possible due to the possibility to add new user-defined primitives – defined through our relDSL – like the SDF primitive in Listing 9. To address this issue, in the future, we plan to use this encoding only for the infinite automata (that motivated it) and provide a simpler more explicit specification language for the finite ones to facilitate their statical analysis.

The principal advantage of our automata representation is that it offers a simple extension mechanism for adding primitive relations. In practice this can be very important for efficiency reasons and can ease the specification of some complex interactions. Besides, some engineers are more familiar to automaton-based specifications (which are more operational) than to their declarative counter-parts.

5.2 Concrete Syntax and Interchange Format

One of the core motivations behind CLOCKSystem is to provide an easy to use, read, and understand syntax for specifying executable time specification inspired by the CCSL logical clock formalism. Hence, it is important to clarify its syntax, and provide a standard mean for model interchange between different environments supporting this formalism (currently CLOCKSystem and TimeSquare). In this section, we first introduce a simple generic syntax for expression CLOCKSystem programs, that also serves as a basis for interoperability. Then we show how using standard Smalltalk messages we can define different problem-domain specific syntactic synonyms that, as we have seen in Sec. 3, renders the CLOCKSystem specifications very short and readable.

Listing 10: Core CLOCKSystem syntax in BNF.

```
system ::= systemDecl
           clockDecl+
           relOrExpDecl+
           yourself
systemDecl ::= "("
           "ClockSystem" "named:" systemName ")"
clockDecl ::= (oneClock | manyClocks) ";"
oneClock ::=
           ("clock:" | "internalClock:") clockName
manyClocks ::=
           ("clocks:" | "internalClocks:") clockList
yourself ::= "yourself" "."
relOrExpDecl ::= "library:" libraryName
           ("relation:" | "expression:") operatorName
           "clocks:" clockList
           ["constants:" constantList]
           ["variables:" varList] ";"
```

```
clockList ::= "#(" clockName+ ")"
constantList ::= "#(" value+ ")"
varList ::= "#(" value+ ")"

xName ::= "#" character+ // Smalltalk symbol
value ::= Object // any Smalltalk object
```

Listing 10 show the BNF specification of the concrete syntax used in CLOCKSystem for the instantiation of the *Clocks* and *ClockRelations* introduced in the last section. The principal characteristic of this syntax is that it is used indiscriminately to instantiate standard CCSL relations (defined in a Kernel library) or to instantiate the user-specific extensions. All these specifications starts by creating a *ClockSystem* object sending the `#named:` message to the ClockSystem class with a String or Symbol as argument, then this object acts as a builder for instantiating *Clock* objects and *ClockRelation* objects. The building of the specification relies on Smalltalk message cascading operator ";". The clocks are instantiated either one by one, or in batch by sending the `#clock:` or `#clocks:` message to the builder (the `internalClock(s):` messages are used for creating intermediate clocks needed by the CCSL expressions). Once the clock declared, the `#library:relation:clocks:constants:variables:` or `#library:expression:clocks:constants:variables:` message is used to instantiate a relation (expression) defined in a given library. To simplify the specification for relations/expressions, that do not need constants and/or variables, for both these messages we define variants rendering the specification of the constant and/or variable lists optional.

In Listing 11 we show the specification of the example introduced in Fig. 1 using this syntax. While still quite readable, this syntax obfuscates somehow the model by: *a)* encoding the clocks, constants and variables as lists; *b)* inlining all constants and variables needed; *c)* making mandatory the specification of the library and relation clauses.

Listing 11: Example of the core syntax encoding the SDF example in Fig. 1 using the relation in Fig. 9

```
(ClockSystem named: #SDF_ex1)
    clocks: #(A B C);
    library: #SDF relation: #channel
        clocks: #(A B)
        constants: #(2 1 -1)
        variables: #(0);
    library: #SDf relation: #channel
        clocks: #(B C)
        constants: #(1 2 -1)
        variables: #(0);
    library: #SDF relation: #channel
        clocks: #(C B)
        constants: #(2 1 -1)
        variables: #(2);
```

Keyword Synonyms. The syntax defined in Listing 10 is simple and generic, however it fails to deliver a short and readable syntax for CLOCKSystem specifications, see Listing 11, nevertheless it is the basis used in our system. To achieve the results presented in Sec. 3 we rely on the definition of "synonym" messages for instantiating the relations or expressions needed. Listing 12 shows the definition of 4 such synonyms for the strict precedence relation. The first one uses the keyword notation used by TimeSquare, the second one uses the standard abstract notation <, while the third innovates by defining the inverse of the < relation (its antonym actually), which can also be interpreted as clock a follows the clock b, which corresponds to our forth synonym message.

Listing 12: Declaring syntactic synonyms for $a < b$ relation

```
Clock>>#precedes: anotherClock
    self system
        relation: #strictPrecedence
        clocks: { self. anotherClock }

Clock >>#< anotherClock
    self precedes: anotherClock
Clock >>#> anotherClock
    anotherClock precedes: self
Clock >>#follows: anotherClock
    self > anotherClock
```

With these mechanisms in place we consider that the embedding has rather succeeded. However, one detail has been overlooked. When offering the support for user-defined syntax one risk is that instead of facilitating communication, the use of syntactic synonyms can hinder it. For example, imagine a specification written with the keywords in another language (it can be pretty difficult to understand). To solve this problem, one solution would be to de-sugar the CLOCKSystem specifications to a standard format, for example the language used by TimeSquare. However, in the case of user-defined "primitive" relations this approach fails. Nevertheless, in CLOCKSystem we do de-sugar the specifications to the rather verbose but generic language presented in Listing 10. In the future, we consider building an ontology of synonyms representing the relations between the message symbols and the CLOCKSystem concepts represented by them, and then de-sugar any specification to a user defined unambiguous set of concepts from this ontology, defaulting to the "core" syntax only for the missing names.

5.3 Execution Semantics and Verification

The execution of logical time specifications, such as ClockSystem, produces series of event occurrences (ticks, instants) that satisfy the constraints imposed by the specified clock relations. These series of events can be seen as a partial order of firings of the clocks involved in the specification. The ticks can be interpreted as the logical activation of some behavior, eg. a processor cycle, activating the computation of the next instruction, or the occurrence of a particular message-send. Thereof, the notion of time captured is decoupled from the physical time and represents essentially notions of coincidence (an event arrives at the same time as another one) and precedence (an event occurs before another

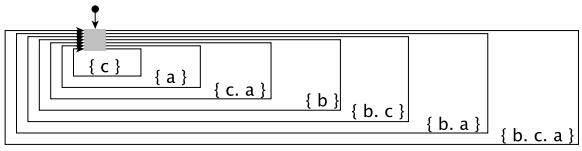


Figure 6: All possible behaviors of a specification with 3 independent clocks

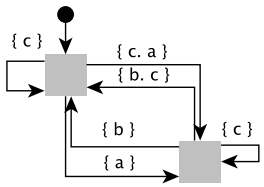


Figure 7: All possible behaviors of a specification with 3 clocks (a, b, c) where the ticks of a alternate with the ones of b.

one) which correspond to the logical view of time introduced by L. Lamport in [14].

Clock Behavior. A single non-constrained clock can be seen as a cyclic infinite behavior that either ticks or does not tick at any given execution step, in other words the clock is free to tick at will. If now we consider the behavior of two or more unrelated clocks together, the expected behavior is that each clock can decide to tick or not to tick on its own (non-deterministically) at any execution step, hence creating an execution sequence containing three possible instant configurations: *1)* only one clock ticks alone; *2)* all clocks tick at the same time, in which case we say that their ticks coincide; *3)* some clocks tick together while others don't. The set of possible behaviors for a system with 3 independent clocks is presented in Fig. 6 as a a Labelled Transition System (LTS), where the labels are synchronisation vectors, as introduced in [2], representing coincident instants of the 3 clocks. This LTS represents all possible execution steps involving the simultaneous tickings of 1, 2, and 3 clocks. Note that this figure is also a complete automaton for which we have represented only the steps with clocks ticking, and that there are implicit transitions that do imply that no clock ticks.

The Impact of Constraints. Adding constraints to such a system reduces the number of possible behaviors to the ones globally allowed by the synchronous parallel composition of the clock behaviors with the constraint behaviors. Fig. 7 shows the emerging behavior of a previously considered 3 clock system, where two clocks are constrained to alternate, and we can see that, for example, the three clocks are not allowed to tick at the same time anymore (the transition labelled {`b. c. a`} in Fig. 6 is not present in Fig. 7).

It is interesting to see that, even though the last two illustrations represent the set of emergent behaviors of a CLOCKSystem specification, graphically they are similar with the primitive constraint automata, shown in Fig. 3 and Fig. 5. This similarity is not incidental, and emerges naturally from the formalism used to represent the CLOCKSystem relations. Mainly, the overall composition of the individual constraints produces an automaton that it is itself a CLOCKSystem constraint. Thus, it can be seen as a complex primitive relation, which can be instantiated as such.

Arnold-Nivat Processes and Verification. The formalism used by CLOCKSystem, introduced and formally defined in [2], and known in the literature as the Arnold-Nivat processes, explicitly expresses the interactions between processes (eg. synchronisation) through a high-level abstraction mechanism, named synchronisation vectors. This mechanism either forces or forbids the simultaneous (coincidence in CCSL parlance) occurrence of a set of events (clocks ticks in our case), which is explicitly defined as tuples labelling the transition relations in a given process (automaton) – ex. { `b. c. a.` } in Fig. 6 is such a tuple. This technique together with a synchronous product operator (also known as synchronous composition of processes) offers a very general and elegant formalism well adapted for our purposes. Moreover, the process of constructing the synchronous product unravels all the reachable states of the system that enables the verification of temporal logic properties (safety and liveness) on the resulting LTS, through a technique known as model-checking [3].

However, in the case of CCSL, due to the presence of infinite clock relations, the construction of the synchronous product cannot be achieved if the combination of constraints does not bound the infinite behaviors. While theoretically problematic – the termination of the composition operation cannot be guaranteed –, in practice the occurrence of infinite behaviors is considered more likely to be a bug than a feature. Hence it is important to statically decide if all infinite relations are bounded, which is turn is a very challenging problem, partially addressed in [22].

Moreover, in some cases, even if the parallel composition pseudo-algorithm can theoretically terminate (finite state-space), in practice we can encounter a state-space explosion problem due to the exponential growth in the number of emerging behaviors of the system with respect to the number of interacting processes (relations in our case), a hard problem that challenges the scalability of computing the exhaustive set of reachable states. Nevertheless, for some types of systems (ex. protocols, control-intensive application, etc.) the possibility to formally verify safety and liveness properties through model-checking relieves the system designer for the burden of testing, and delivers strong guarantees to

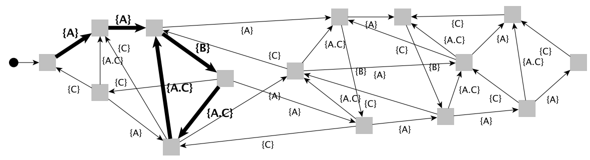


Figure 8: Exhaustive reachability analysis result for the SDF example with channel capacity bounded at 4 tokens.

the system users. To address these cases the CLOCKSystem toolkit supports the parallel composition of clock relations through a pseudo-algorithms, similar to the one introduced in [22]. This pseudo-algorithm is implemented in Smalltalk using the BuDDy BDD[4] package [16] for clock assignment resolution.

Fig. 8 shows the exhaustive state-space exploration results, obtained with CLOCKSystem, for the SDF example introduced in Fig. 1. In this case the channel capacity of each channel was bounded at 4 tokens to ensure a finite state-space[5]. This result represents all the execution paths (sequences of clock tickings) allowed by the specification, and amongst them we can identify the cyclic trace presented in Fig. 2a (emphasised with bold lines).

Traces and Simulation. To alleviate all these complications another well known technique can be used to prove the presence of property violations, instead of their absence. This technique, commonly known as simulation, extracts particular executions traces from the set of possible behaviors by walking through the LTS automaton of the composition either explicitly or implicitly. In a practical setting extracting a trace explicitly does not solve the previous issues since the LTS should be constructed first, however can prove very useful for understanding and debugging parallel composition results. One particular execution trace can also be extracted dynamically (implicitly) during the process of parallel composition by simply choosing one and only one transition to execute from the set of alternatives possible at any given execution point. The decision procedure used to select the transition to fire can rely on any heuristic decision process, in the context of CCSL a number of such heuristics were proposed in [1] and are currently implemented in TimeSquare and CLOCKSystem.

In terms of simulation facilities, as opposed to TimeSquare which implements a simulator by the direct interpretation of the CCSL operational semantics (providing only trace extractions implicitly), the CLOCKSystem simulator relies on the parallel composition of automata and offers the possibility to use either the explicit or the implicit trace extraction techniques. In CLOCKSystem , the extracted execution traces are in fact just a subgraph of the resulting LTS graph. Which can be either interpreted for a finite number of steps or fed as input of other analysis tools. One example of such a trace is presented in Fig. 2a, with its interpretation for 21 observable simulation steps in Fig. 2c. Note that our interpreter ignores the eventual invisible steps (the ones without ticking clocks). Also note that through our encoding these traces could be also interpreted as execution contexts, for integration with other verification approaches such as Context-aware Verification [8].

5.4 Practical considerations

CLOCKSystem was implemented in Pharo Smalltalk version 3. For the implementation of the synchronous parallel composition of automata we rely on the use of BuDDy BDD library [16] linked and used from the Smalltalk image through the high-performance NativeBoost FFI interface [5]. We implemented a simple tri-state logic solver in Smalltalk which can be used for the platforms where the BDD library is not available. A simple editor for ClockSystem specifications was developed using the Glamour toolkit, and the Roassal framework was used visualisations [4].

6. Conclusion and Perspectives

In the context where our execution platforms are becoming complex distributed systems on a chip, by integrating more and more heterogenous computing resources (processor cores, graphical accelerators, etc) the need for time-driven reasoning becomes a necessity for software systems in general. CLOCKSystem addresses the lack of support for reasoning about time and its implications in general-purpose programming languages. While, currently the CLOCKSystem and the associated tools are in their infancy, we believe that our logical time embedding in Smalltalk already promises a symbiotic relation with its host environment.

In this study we have presented CLOCKSystem, an embedding of a logical representation of time into the Pharo Smalltalk environment. This environment re-uses concepts from the CCSL formalism, which was adopted for the formalisation of time specifications in the UML Marte environment, and extends this formalism by adding the possibility to define new primitive "clock relations" through an automata-based approach. Moreover, the CLOCKSystem language borrows the syntax of CCSL, for which it builds an DSL embedded in Smalltalk through the usage of message-sends and relations synonyms. By presenting a case-study encoding the control aspects of Synchronous Data-Flow applications, this DSL was compared to the abstract and TimeSquare specifications and was shown to be readable and very close to the abstract notation of CCSL. The importance of the contribution was emphasised through five usage scenarios that are enabled by the CLOCKSystem toolkit. And finally some of the implementations details were discussed, a generic interchange format was proposed, and some principles of the

[4] BDD – Binary Decision Diagram

[5] Note that the capacity bound – 4 – was chosen arbitrarily and any bound > 2 would have produced similar results but with a smaller state-space for 2 and 3 and larger state-space for any value > 4.

CLOCKSystem execution semantics were briefly presented emphasising some of the difficulties of the formalism.

Future research directions include: *a)* improving the support for statically detecting if the constraint system is bounded (finite state-space); *b)* extending the expressive power of CLOCKSystem by integrating support for dense-time representations, inspired by timed-automata formalisms; *c)* integration mechanisms for reasoning about dynamic environments, where the "clock" are dynamically created during the lifetime of the application; *d)* studying the potential incidence of CLOCKSystem constraints and execution traces can have for state-space decomposition in model-checking.

Acknowledgments

The author would like to thank Zoe Drey for kindly reviewing early drafts of this paper. Equally, we acknowledge the fruitful discussions with Joel Champeau, Luka Le Roux, Jean-Charles Roger, and Philippe Dhaussy that led to the results presented in this work.

References

[1] C. André. Syntax and Semantics of the Clock Constraint Specification Language (CCSL). Rapport de recherche RR-6925, INRIA, 2009. URL http://hal.inria.fr/inria-00384077.

[2] A. Arnold. Synchronized products of transition systems and their analysis. In J. Desel and M. Silva, editors, *Application and Theory of Petri Nets 1998*, volume 1420 of *Lecture Notes in Computer Science*, pages 26–27. Springer Berlin Heidelberg, 1998. ISBN 978-3-540-64677-8. . URL http://dx.doi.org/10.1007/3-540-69108-1_2.

[3] C. Baier and J.-P. Katoen. *Principles of Model Checking*. Representation and Mind. The MIT Press, 2008. ISBN 026202649X, 9780262026499.

[4] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. *Pharo by Example*. Square Bracket Associates, 2009. ISBN 978-3-9523341-4-0. URL http://pharobyexample.org.

[5] C. Bruni, S. Ducasse, I. Stasenko, and L. Fabresse. Language-side Foreign Function Interfaces with NativeBoost. In *International Workshop on Smalltalk Technologies*, Annecy, France, Sept. 2013. URL http://hal.inria.fr/hal-00840781.

[6] J. Deantoni and F. Mallet. ECL: the Event Constraint Language, an Extension of OCL with Events. Research Report RR-8031, INRIA, July 2012. URL http://hal.inria.fr/hal-00721169.

[7] J. DeAntoni and F. Mallet. Timesquare: Treat your models with logical time. In C. Furia and S. Nanz, editors, *Objects, Models, Components, Patterns*, volume 7304 of *Lecture Notes in Computer Science*, pages 34–41. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-30560-3. . URL http://dx.doi.org/10.1007/978-3-642-30561-0_4.

[8] P. Dhaussy, J.-C. Roger, L. Leroux, L. E. Bretagne, B. France, and F. Boniol. Context aware model exploration with obp tool to improve model-checking. *Embedded Real-Time Software and Systems (ERTS'12)*, 2012.

[9] M. Eysholdt and H. Behrens. Xtext: Implement your language faster than the quick and dirty way. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, SPLASH '10, pages 307–309, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0240-1. . URL http://doi.acm.org/10.1145/1869542.1869625.

[10] P. Farail, P. Gaufillet, F. Peres, J.-P. Bodeveix, M. Filali, B. Berthomieu, S. Rodrigo, F. Vernadat, H. Garavel, and F. Lang. FIACRE: an intermediate language for model verification in the TOPCASED environment. In *European Congress on Embedded Real-Time Software (ERTS)*, Toulouse, january 2008. SEE.

[11] P. L. Guernic, J.-P. Talpin, and J.-C. L. Lann. Polychrony for system design. *Journal of Circuits, Systems, and Computers*, 12(3):261–304, 2003.

[12] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, Sep 1991. ISSN 0018-9219. .

[13] G. Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.

[14] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978. ISSN 0001-0782. . URL http://doi.acm.org/10.1145/359545.359563.

[15] P. LeGuernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with signal. *Proceedings of the IEEE*, 79(9):1321–1336, Sep 1991. ISSN 0018-9219. .

[16] J. Lind-Nielsen. BuDDy: A Binary Decision Diagram library. http://buddy.sourceforge.net.

[17] F. Mallet. Automatic generation of observers from marte/ccsl. In *Rapid System Prototyping (RSP), 2012 23rd IEEE International Symposium on*, pages 86–92, Oct 2012. .

[18] F. Mallet, J. DeAntoni, C. André, and R. de Simone. The clock constraint specification language for building timed causality models. *Innovations in Systems and Software Engineering*, 6(1-2):99–106, 2010. ISSN 1614-5046. . URL http://dx.doi.org/10.1007/s11334-009-0109-0.

[19] N. Menad and P. Dhaussy. A transformation approach for multiform time requirements. In R. Hierons, M. Merayo, and M. Bravetti, editors, *Software Engineering and Formal Methods*, volume 8137 of *Lecture Notes in Computer Science*, pages 16–30. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-40560-0. . URL http://dx.doi.org/10.1007/978-3-642-40561-7_2.

[20] OMG. Uml profile for marte: Modeling and analysis of real-time embedded systems, 2009.

[21] D. Pilaud and N. Halbwachs. From a synchronous declarative language to a temporal logic dealing with multiform time. In M. Joseph, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 331 of *Lecture Notes in Computer*

Science, pages 99–110. Springer Berlin Heidelberg, 1988. ISBN 978-3-540-50302-6. . URL http://dx.doi.org/10.1007/3-540-50302-1_5.

[22] Y. Romenska and F. Mallet. Improving the efficiency of synchronized product with infinite transition systems. In V. Ermolayev, H. Mayr, M. Nikitchenko, A. Spivakovsky, and G. Zholtkevych, editors, *Information and Communication Technologies in Education, Research, and Industrial Applications*, volume 412 of *Communications in Computer and Information Science*, pages 285–307. Springer International Publishing, 2013. ISBN 978-3-319-03997-8. . URL http://dx.doi.org/10.1007/978-3-319-03998-5_15.

[23] Y. Romenska and F. Mallet. Lazy parallel synchronous composition of in finite transition systems. In *International Conference on ICT in Education, Research and Industrial Applications*, volume 1000, pages 130–145, Kherson, Ukraine, June 2013. CEUR-WS.org.

[24] F. Schreiber. Is time a real time? an overview of time ontology in informatics. In W. Halang and A. Stoyenko, editors, *Real Time Computing*, volume 127 of *NATO ASI Series*, pages 283–307. Springer Berlin Heidelberg, 1994. ISBN 978-3-642-88051-3. . URL http://dx.doi.org/10.1007/978-3-642-88049-0_14.

[25] R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7(3):149–174, 1994. ISSN 0178-2770. . URL http://dx.doi.org/10.1007/BF02277859.

[26] L. Yin, F. Mallet, and J. Liu. Verification of marte/ccsl time requirements in promela/spin. In *Engineering of Complex Computer Systems (ICECCS), 2011 16th IEEE International Conference on*, pages 65–74, April 2011. .

From Smalltalk to Silicon: Towards a methodology to turn Smalltalk code into FPGA

LE Xuan Sang[1,2], Loïc Lagadec[1], Luc Fabresse[2], Jannik Laval[2] and Noury Bouraqadi[2]

[1]Lab-STICC, ENSTA Bretagne
[2]Institut Mines-Telecom, Mines Douai

Due to their ability to combine high performances along with flexibility, FPGAs (Field Programmable Gate Array) are used in robotic applications nowadays, especially in case of real-time applications. The FPGA circuits are often designed and configured using the Hardware Description Languages (HDLs) like VHDL or Verilog. However, although these languages provide abstractions up to the functionality level, they lack many features of todays modern languages that make them unsuited for high-level models and systems. In this paper, we present an overview of a methodology that uses a Dynamic Reflective Language, such as Smalltalk, for high level hardware/software co-design on FPGAs.

***Index Terms*—Smalltalk, Pharo, FPGA, VHDL, Native Boost, robotic, Dynamic Reflective Language.**

I. INTRODUCTION

A fundamental robotic application often consists of three general states: (1) *perception* in which the robot senses and analyses the environment via its sensors. (2) *Planification* that helps the robot to take decision based on its sensation. (3) The *Control* state is the reaction of the robot where its planification takes effect on the actuators to answer the changes of environment. A typically example is given by a robot with a vision system consisting of a fixed camera, which takes pictures of a scene, recognises an object, identifies its location, calculates the trajectory and commands the actuators to follow the object. The more sensors the robot has, the better its ability to interact with the environment and to guarantee a stable behaviour as well as predictable performance. However, this will cause two major problems: first, multi-sensor processing and analysing demand a signifiant processing power, especially in case of real-time applications which have a heavy response time constraint. Second, adding more sensors may change the hardware configuration of the system and thus can require the replacement of other devices consequently which will raise the production cost.

On the software side, to improve the productivity, we use Smalltalk [1], [2], a high level dynamic language, in the development task. With its simplicity and rich semantic, the language makes the programmation task significantly faster and simpler. However, Smalltalk is not very suitable for mass datas processing (multi-sensor datas), especially in the context of a real-time application which requires the parallel processing of multi-data sources.

These hardware/software challenges of flexibility and performance can be overcome by using FPGAs. FPGAs are integrated circuits which contain a matrix of reconfigurable gate array (logic block) that, when configured, implement a circuit [3], [4]. FPGA circuits use hardware for processing logic and thus may not depend on any operating system. Because the processing paths are parallel, different operations do not have to compete for the same processing resources. That is, the operational speed can be very fast [5]. The reconfigurability of FPGAs is another interesting point of this kind of hardware which turns it to a limitless flexible device. This ability provides designers with a way to make different hardware configurations on the same chip. This means that, each program that uses a FPGA chip can download a new circuit design onto the chip and tailor it specifically for the needs of that program. With these abilities, FPGA is a powerful and relatively inexpensive solution which responds to the demand of high processing power and flexibility to the unforeseen change of hardware configuration in the robot application.

To avoid confusion, in this paper, we make a convention that the term **design** is used only for the *digital hardware design task* (section II) that implements the FPGA circuits. For the *robotic software development* on FPGA (section III), we use the term **program/programing** instead.

The FPGAs circuits are often designed using a *Hardware Description Language* like VHDL or Verilog [6], [4]. The HDLs have been evolved in recent years. They provide a simpler approach to digital hardware design. But in comparison with today modern software technique, this evolution is quietly not enough. These HDLs allow the specification at the *Register Transfer Level* (RTL). But at algorithmic level, their lack of semantics makes behavioural verification and debugging hard. Moreover, since these languages are specific for hardware description, they are not adequate for high-level models or systems which require a hardware/software co-design. An HDL-based design is often constrained on some target FPGAs that limits its reusability on the others. This slows down the production of new designs and makes difficult to maintain or extend existing designs. A high-level language like Smalltalk provides all the features that the HDLs miss. Furthermore, it offres also a valuable ability of debugging, testing and probing application which is very useful for the behavioural verification of the hardware design. A FPGA HDL-based design has many similarities with software development and therefore can be modelled by using a high-level language. There are some works which are targeting on this, such as

[7], [8], [9], [10], using C++, Java or Python. The well-known SystemC [11], [12], [13] is a typically example, by extending the mainstream C++, it provides a class library that enables to describe and simulate software/hardware at system level. This brings the advantages of oriented-object software development to the digital hardware design world.

This paper will attach on two sides of the hardware/software co-design problem on FPGA. We first present an overview of a hardware design methodology that uses Smalltalk as a hardware description and verification language. Here we propose a modelling methodology that acts as an abstraction layer between Smalltalk and VHDL, this layer produces the VHDL code from Smalltalk code and invokes the vendor's synthesis toolchain to actually make the code available on the FPGA (section II). Secondly, we aim at a software development approach that allow us, by using Smalltalk, to program the robotic software that need interact with the FPGA. This section presents also the FPGA-ARM System on Module (SoM) or System on Chip (SoC [14]) which gives us the ability to interact directly to the FPGA's registers via system libraries (section III). At the end of the paper (section IV), we discuss what we have done so far and the future works based on the methodologies proposed.

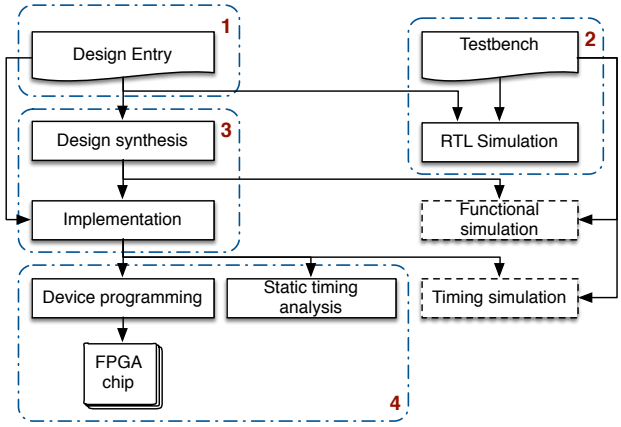


Figure 1: A HDL-based development flow: (1) Design the system and derive the HDL files. (2) Write the testbench and perform a RTL simulation to verify the design. (3) Synthesis and implement the design using the vendor's toolchain. (4) Download the binary file (proprietary format) onto FPGA memory. The *functional simulation* and *the timing simulation* are optional and thus can be omitted from the development flow.

II. A GENERAL MODELLING METHODOLOGY TO DESIGN FPGA CIRCUITS USING SMALLTALK

This section presents the FPGA hardware design side, in which we use Smalltalk as a high level hardware description an verification language.

A. *Background : FPGA concepts and development life-cycle*

The FPGA circuits, in general term, are often designed using a HDL language (such as VHDL or Verilog) and configured by the vendor's toolchain. Figure 1 shows the development flow of a FPGA HDL-based design [4]. The left portion represents the design and refinement process in which the design is transformed from an abstract HDL description to a device cell-logic configuration before being downloaded onto the chip. The right portion is the design verification process to check that the design is correct (RTL Simulation) and meets the functionality requirements (functional simulation) as well as the performance constraints (timing simulation).

The *synthesis* is known as the logic synthesis, in which the HDLs is transformed from the RTL constructs to generic gate level components. The *implementation* process consists of three smaller sub-processes : The *translate* process merges the designs to a single netlist. The *map* process maps the generic gates in the netlist to FPGA's logic cells. Finally the *place and route process* defines the physical layout inside the FPGA and connects the logic blocks together. These two processes strictly depend on the vendor's tool (Xilinx, Altera etc.).

B. *Smalltalk as a hardware description an verification language*

As mentioned before, the lower steps (marked as 3 and 4 in figure 1) are strongly coupled to the vendor's tools and therefore are difficult to change. Our proposition is to model only the top level of the flow, concretely, the *HDL design task* and the *RTL Simulation task* (marked as 1 and 2). The main objective is to be able to use Smalltalk as a high-level hardware description and verification language; as well as to benefit from its integrated development environment to debug and manage the hardware designs.

The overall architecture of our methodology is shown in figure 2. First we need to build an *Hardware design abstraction layer* that acts as a abstraction layer between Smalltalk and VHDL. This layer models the basic principles of the VHDL, and thus can turn Smalltalk to a high-level hardware description language. It will handle all of ours *Smalltalk-based design entries* on the top.

From this abstraction layer, two lower modules will be developed : (1) one provides the Smalltalk-VHDL conversion ability which can help to generate an HDL-based design entry from the Smalltalk's one. This module will create a path to the traditional design flow that brings our design to the real hardware via the vendor's toolchain. (2) The other handles the *RTL Simulation task* where we can verify the correctness of our design right inside the Smalltalk environment. Note that, for the lower simulation levels such as *functional simulation* and *timing simulation*, we need the vendor's tools consequently.

Since our purpose is to propose a simple and more efficient way to design the FPGA circuits, the methodology described above must meet some requirements below:

Correctness of conversion: The Smalltalk-to-VHDL conversion must ensure the preservability of the algorithm. Moreover, since the synthesis and implementation are done automatically behind the scene, the produced VHDL must be a already-synthesizable VHDL without any further modifica-

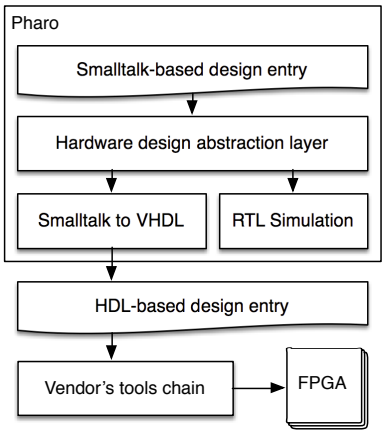


Figure 2: A Smalltalk-based development flow: the modelling is done on top, at the language and simulation level and then provides a path to the vendor's tools to make the design available on the real FPGA.

tion on it.

Reusability and Extensibility: The abstraction layer must handle the compatibility of the Smalltalk-based design with different FPGA devices and make it hardware-independent. This allows the reusability of the design without (or with a minimal) effort of modification. The design will also need to be extensible by subclassing, the more it is subclassed, the more its functionalities are enriched.

Simulation: As we only work at RTL level, the simulation must allow the designer to verify the correctness of the design at that level. It's ideal that the simulator can support the waveform by tracing the signal changes in a VCD (Value Change Dump) file. This kind of file can be viewed using an external viewer like GTKWave[1], or much better, one developed natively in Smalltalk [15], [16]. The possibility of using Unit test (SUnit) in the hardware design is also very appreciated.

Robust interaction with vendor's tools: There are different FPGA vendors out there (Xilinx, Altera, etc.), and every vendor has many FPGA families different (Xilinx: Virtex, Spartan, Artix, etc.). Therefore, in order to configure the design on any FPGA chip, we need to provide its hardware description to the vendor's toolchain. The methodology must propose an efficient way to facilitate and automate this process [17], [18].

III. SOFTWARE PROGRAMMING ON FPGAs WITH SMALLTALK

This section describes the approaches to communicate with the FPGA circuits from our Smalltalk-based robotic application.

A. *Standalone FPGA*

A standalone FPGA is a FPGA which can independently operate side by side with the host system. This FPGA has pre-configured circuits on it in order to perform a fixed algorithm (image filter for example). Figure 3 shows a typically communication flow between the Smalltalk-based software and the FPGA's circuits via a common hardware interface like USB,RS232,etc. Here we use a Foreign Function Interface (FFI) such as NativeBoost[19], [20] to get access to the C librairies which define the interaction protocol with the FPGA circuits.

Although this approach is simple and easy to implement, it presents a potential risk of bottleneck when using these communication interfaces. In fact, the processing time on FPGA is fast, but the data transfer between the host system and the FPGA may be costly and therefore, can drop down the performance of the overall system.

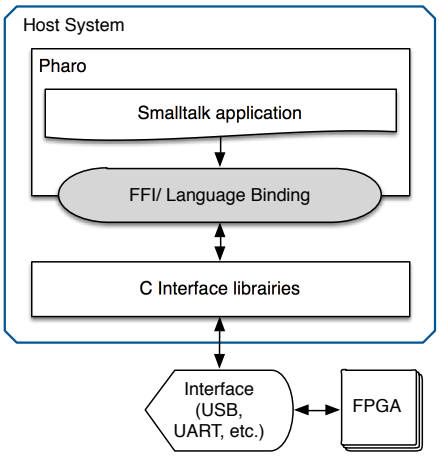


Figure 3: A Smalltalk-based communication flow on a standalone FPGA: the application on the host system interacts with the FPGA's circuits via an interface of communication such as USB,RS232, parallel, etc.

B. *Optimisation of software/hardware interaction with FPGA-ARM SoM/SoC*

A FPGA-ARM System on Module (SoM) or System on Chip (SoC) is an integrated circuit that integrates an ARM-based hard processor system (consisting of processor, peripherals and memory interfaces) with a FPGA chip into a single module/chip. This integration brings Linux on top of the system as the software layer and makes the communication between FPGA and ARM more efficient while simplifying the development. Software application can talk to FPGA via its Extension Processing Platform Architecture [21] (usually provided by vendor) which allows us to interact directly with the FPGA registers. This will reduce the bottleneck problem.

For this kind of FPGA-systems, we introduce a way to communicate with the FPGA by accessing to its registers. As shown in the figure 4, we host the Pharo Smalltalk on top of the embedded system (there is already a virtual machine for the ARM[2] architecture), and then build a Smalltalk abstraction layer to make our Smalltalk code talks to the FPGA registers via system librairies. At this point, the *Register interaction abstraction layer* provides a path to the

[1]http://gtkwave.sourceforge.net

[2]https://ci.inria.fr/pharo-contribution/view/ARM/

system librairies (Drivers/ Extension Processing Plateforme architecture) and brings their functionalities to the Smalltalk environment which opens a way to interact directly with FPGA registers from our Smalltalk application.

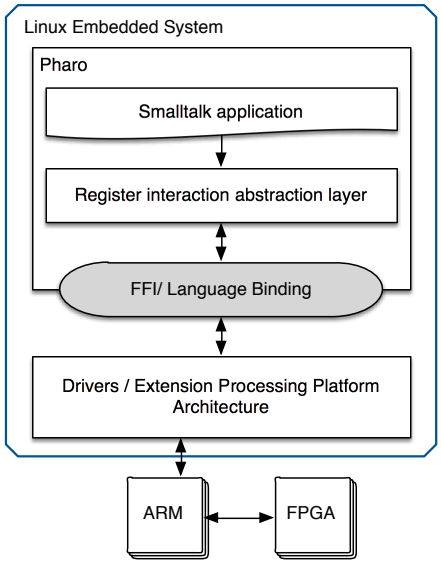


Figure 4: A Smalltalk-based communication flow on a FPGA-ARM OMS/SoC : we use FFI (Native Boost for example) to access to system libraries in order to interact with FPGA circuits registers.

Note that, to program the system by this way, a pre-configured circuit is required to be available on the FPGA chip which provides the registers that our software want to interact with.

IV. EXPERIMENTAL VALIDATION

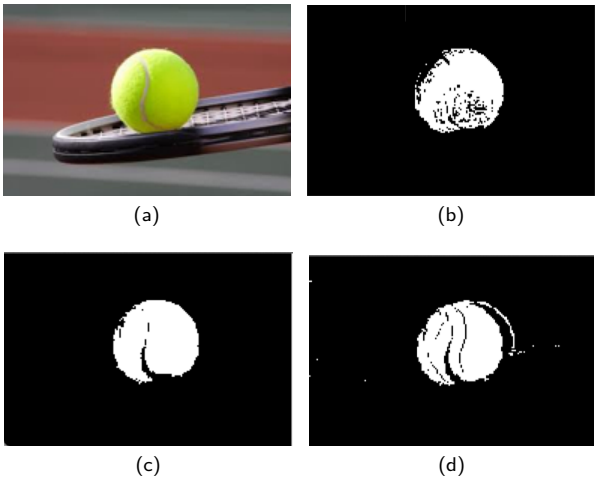
To make a proof of the challenges presented in section I, we first build a reference real-time robotic application fully in Pharo Smalltalk using an actual robot. By analysing this one, we would like to define the critical parts that have a negative impact on its performance. These parts will then be projected (designed) on FPGA to obtain a significant optimisation. Obviously, the transformed application will be evaluated quantitatively, and the expected result is that we'll win an important gain in term of performance when using FPGA.

For the reference robotic application [22], we chose to develop a simple object tracking system (by color pattern) with a wheeled robot [23] available at Institut Mines-Telecom (as shown in figure 5): the robot uses its camera to scan the whole environment and sends back an image stream. The application filters each received image by a color pattern using a HSV filter and looks for the target object (for example, a tennis ball). If the object is found, the laser sensor data will be collected from the robot to mesure the distance to the object. Based on these parameters, the application will command the robot to move (forward, backward, rotate, etc) such way that the robot maintains always a safe (constant) distance with the object. The application has been entirely developed in Smalltalk with the help of PhaROS [3], a Pharo package that allows us to interact with the robot via the ROS [4] middleware [24], [25].

Figure 5: The Robulab.

When testing with the camera resolution of 320x240 (32 bit image, lowest resolution), we found that the application took around 230 ms to completely process each image, meaning 4 images per second. This speed is obviously far away from a real time application which demand at least 15-20 images per second. This is the critical part that need to be accelerated through using FPGA.

We are currently working on the projection of the image processing part on FPGA to obtain a hardware version of the algorithm. A first performance comparison between software and hardware implementation of the HSV filter [26], [27] has been performed. The algorithm gets a RGB image as input, transforms it to HSV color space and then filters it by a specific color pattern. We've implemented 3 versions of this algorithm using Pharo Smalltalk, C (with OpenCV) and FPGA circuits.



(a) (b) (c) (d)

Figure 6: HSV filtre : (a) Original image; images filtered using Pharo (b), openCV (c) and FPGA circuits (d)

Figure 6 shows the experimental results of these ones on an 192x128 image, 32 bit with the color pattern: $75 \leq H \leq$

[3] http://car.mines-douai.fr/2014/02/pharos-fosdem-2014-slides/
[4] Robot Operating System

150, $0.3 \leq S \leq 1.0$, $0.5 \leq V \leq 1.0$ (the tennis ball color range). Although there are a slightly difference between the filtered images, the object, in general term, is well identified in all case. On the processing performance side, we found that to completely filter the image, the Smalltalk, C, and FPGA implementation took around 73, 1.5 and 2.5 milliseconds respectively. That is, when passing from Smalltalk code to FPGA circuits, we obtain a very important gain (about 97%) in term of performance. However, the C implementation (with openCV) is shown faster than the FPGA one (around 1 millisecond) which seems surprising theoretically. In fact, in this experiment, we use a standalone FPGA and the image is transferred between the host system and the FPGA via an USB connection [28] for filtering. Here we encounter the bottleneck problem which drops down the circuits performance. With a FPGA-ARM SoC/SoM, this problem can be optimised and the FPGA implementation can perform more efficiently.

V. CONCLUSION

This paper presents the state of the art of an approach to use Smalltalk for software/hardware co-design on FPGAs. In this work, we focus mainly on a general modelling methodology of hardware design at the behavioural (high abstraction) level that can make the hardware design task simpler and more efficient. We also propose a theoretical Smalltalk-based solution to communicate with the FPGA circuits (Standalone or SoC/SoM FPGA). We finally show a experimental comparison of performance between software and hardware implementation of a HSV filter algorithm which help us figure out some theoretical hypothesises.

References

[1] A. P. Black, D. Pollet, D. Cassou, and M. Denker, *Pharo by Example*. Square Bracket Associates, Switzerland, 2009.

[2] A. Bergel, D. Cassou, S. Ducasse, and J. Laval, *Deep into Pharo*. Square Bracket Associates, Switzerland, 2013.

[3] R. Robbins, "Advantages of FPGAs," 2010. [Online]. Available: http://www.controlengeurope.com/article/32043/Advantages-of-FPGAs.aspx

[4] P. P.Chu, *FPGA Prototyping by VHDL examples (Xilinx Spartan-3 version)*. John Wiley & Sons, Inc., 2008.

[5] C. Cullinan, C. Wyant, T. Frattesi, and X. Huang, "Computing Performance Benchmarks among CPU , GPU , and FPGA," *WPI*, 2012.

[6] Mike Field, "Introducing the Spartan 3E FPGA and VHDL," 2012.

[7] S. Vernalde, P. Schaumont, and I. Bolsens, "An object oriented programming approach for hardware design," in *VLSI '99. Proceedings. IEEE Computer Society Workshop On*, 1999, pp. 68–73.

[8] T. Kuhn, T. Oppold, M. Winterholer, W. Rosenstiel, M. Edwards, and Y. Kashai, "A framework for object oriented hardware specification, verification, and synthesis," *Proceedings of the 38th conference on Design automation - DAC '01*, pp. 413–418, 2001. [Online]. Available: http://portal.acm.org/citation.cfm?doid=378239.378537

[9] M. Geilen, J. Voeten, P. van der Putten, L. van Bokhoven, and M. Stevens, "Object-oriented modelling and specification using SHE," *Computer Languages*, vol. 27, no. 1-3, pp. 19–38, Apr. 2001. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0096055101000145

[10] J. Decaluwe, "Design hardware with Python." [Online]. Available: http://www.myhdl.org/start/overview.html

[11] S. Swan, "An Introduction to System Level Modeling in SystemC 2 . 0," Cadence Design Systems, Inc., Tech. Rep. May, 2001.

[12] "SystemC." [Online]. Available: http://systemc.org

[13] Synopsys, "Functional specification for SystemC 2.0," 2002.

[14] . Wikipedia english, "System on a chip." [Online]. Available: http://en.wikipedia.org/wiki/System_on_a_chip

[15] D. Picard and L. Lagadec, "Multi-Level Simulation of Heterogeneous Reconfigurable Platforms," *ReCoSoC'08, Barcelona, Spain*, 2008.

[16] L. Lagadec and D. Picard, "Smalltalk debug lives in the matrix," *International Workshop on Smalltalk Technologies on - IWST '10*, pp. 11–16, 2010. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1942790.1942792

[17] S. Guelton, "Building Source-to-Source Compilers for Heterogeneous Targets," Ph.D. dissertation, Université européenne de Bretagne, 2011.

[18] E. M. Panainte, "The Molen compiler for reconfigurable architectures," Ph.D. dissertation, Universitatea Politehnica Bucuresti, 2007. [Online]. Available: http://www.narcis.nl/publication/RecordID/oai:tudelft.nl:uuid:8150156e-e633-4319-8685-ccac7b083434

[19] L. Laffont, S. Ducasse, and I. Stasenko, "NativeBoost Recipes : The X11 Journey."

[20] I. Stasenko, C. Bruni, S. Ducasse, and L. Fabresse, "Language-side Foreign Function Interfaces with NativeBoost," in *Proceedings of International Workshop on Smalltalk Technologies*, 2013.

[21] XilinX, "Zynq-7000 All Programmable Software Developers Guide," 2014.

[22] L. X. Sang, "Programming the Robulab to follow an object using color pattern," Institut Mines-Telecom, Mines Douai/ ENSTA Bretagne, Tech. Rep., 2014.

[23] ROBOSOFT, "Robulab 10 user 's manual," 2008.

[24] A. Marin-Hernandez, "Robot Operation System," RAP-LAAS-CNRS,DAI-Universidad Veracruzana, Tech. Rep., 2012.

[25] A. Martinez and E. Fernández, *Learning ROS for robotics programming*. Birmingham: Packt Publ., 2013.

[26] T. Hamachi, H. Tanabe, and A. Yamawaki, "Development of a Generic RGB to HSV Hardware," *The Proceedings of the 1st International Conference on Industrial Application Engineering 2013*, pp. 169–173, Mar. 2013. [Online]. Available: https://www2.ia-engineers.org/conference/index.php/iciae/iciae2013/paper/view/81

[27] B. M. Krishna, K. G. Deepika, B. R. Kanth, and V. G. S. Vemana, "Article: Image processing using ip core generator through fpga," *International Journal of Computer Applications*, vol. 46, no. 24, pp. 48–52, May 2012, published by Foundation of Computer Science, New York, USA.

[28] M. Chris, "FPGALink User Manual," 2012.

[29] W. Meeus, K. Van Beeck, T. Goedemé, J. Meel, and D. Stroobandt, "An overview of today's high-level synthesis tools," *Design Automation for Embedded Systems*, vol. 16, no. 3, pp. 31–51, Aug. 2012. [Online]. Available: http://link.springer.com/10.1007/s10617-012-9096-8

Live Programming the Lego Mindstorms

Johan Fabry Miguel Campusano
PLEIAD and RyCh Laboratories
Computer Science Department (DCC)
University of Chile
{jfabry,mcampusa}@dcc.uchile.cl

Abstract

Development of software that determines the behavior of robots is typically done in a language that is far from dynamic. Programs are written, compiled, and then deployed on a simulator, or the robot, for testing. This long development cycle causes a cognitive dissociation between writing the code for the robot and observing the robot in action. As a result, writing robot behaviors is much more difficult than it should be. In contrast, live programming proposes an extraordinary tightening of the development cycle, yielding an immediate connection between the program and the resulting behavior. To achieve live programming for robot behaviors, we designed and implemented the LRP language. In this paper we show how LRP interfaces with the Lego Mindstorms EV3, report on experiences programming Lego robots, and discuss how salient features of the language were made possible thanks to its implementation in Pharo Smalltalk.

1. Introduction

The origins of live programming can be traced back to the early work of Tanimoto on Viva [12]. It states that "A live system begins the active feedback at editing time, and then continues it through the remainder of the session or until explicitly disabled by the user." Such live programming allows programmers to benefit from an immediate connection with the program that they are making. This is because the development cycle is extremely tight and there is no cognitive dissociation between writing the code and observing its execution.

In our research we aim to bring the advantages of live programming to programming of robots, more specifically the behavior layer. The behavior layer is the part of the software of the robot that acts on processed inputs to realize specific actions of the robot, *i.e.* its behavior. Typically, such behavior is written in a language that is far from dynamic, compiled, and then deployed on a simulator (or the robot itself) for testing. In this long cycle the cognitive distance between the program and the resulting robot behavior is vast, resulting in a high degree of difficulty of getting these behaviors to work well. For example, it is frequently the case that the programmer observes the robot (or the simulation) performing some specific movement and it is totally unclear *why* this movement is happening. With live robot programming this cognitive distance almost disappears. This is because the development environment includes a visualization of program execution that transparently updates on each program change, in addition to the execution being reflected in the robot simulator or even on the running robot itself.

To allow live programming of robot behaviors we have developed the Live Robot Programming (LRP) language. This language is based on the nested state machine paradigm, as this paradigm has proven to be well-suited to define robot behaviors [8, 14]. LRP is designed from the onset to be a live programming language, and as such comes with its own state machine interpreter and visualization of existing machines. The language is not hardcoded to a specific robot platform, instead relying on bridging software to access specific robot APIs.

In this paper we show how LRP enables live programming of the Lego Mindstorms EV3 robot platform through JetStorm [7], report on our experiences programming the Mindstorms in LRP, and discuss specific points of the implementation of LRP that were facilitated largely by the language features and infrastructure present in Pharo Smalltalk.

This paper is structured as follows: the next section gives a brief overview of the LRP language, using an example

behavior that also serves to illustrate elements of the rest of the paper. Section 3 reports on the bridge to the Mindstorms and our experience in using it to program robot behaviors. Following this, Section 4 highlights specific elements of Pharo Smalltalk that made the implementation possible. The paper then presents related work, future work and concludes.

2. The LRP Language

Live Robot Programming (LRP) is a live programming, nested state machine based language with an associated interpreter and visualization, implemented in Pharo. The features of LRP are designed for robot programming, yet the language is not hardcoded to a specific robot platform. LRP enables the use of APIs of specific robot platforms and as such comes with bridges towards the Robot Operating System (ROS) [5], and now also to the Mindstorms EV3 [13] through JetStorm [7], as will be discussed in Section 3.

A complete description of LRP is outside of the scope of this paper, we only give a brief overview of its features here, and refer to its website `http://pleiad.cl/LRP` and other published work [4] for more details.

The main language features of LRP are:

- *Machines* with states and different kinds of transitions.
- *Transitions* that can occur on events, occur after a timeout or occur automatically after a state is entered.
- *Events* are explicitly defined and trigger if their included piece of code, called an *action*, evaluates to true.
- *States* can have actions that are run when entering the state, leaving the state, or when the state is active.
- States can define state machines, which enables nesting.
- Machines can define variables, and these are accesible inside actions if the variable is lexically in scope.

LRP has its own language syntax and the interpreter is, in essence, a plain state machine interpreter that consumes the ASTs of the program and provides the standard nested state machine semantics. The only remarkable element is that actions are actually Smalltalk blocks that are compiled after the program is parsed. This process is discussed in Section 4.1.

To show the syntax of the language, clarify how it allows for robot programming on the Mindstorms and provide example material for Sections 3 and 4, we now show and discuss the code for a simple behavior. The behavior is a simple space exploring behavior where the robot goes forward until it encounters a wall, where it backs up, turns a bit, and again goes forward. This behavior is ment to run on a differential-drive robot[1] with the (ultrasonic) distance sensor pointing forward and a touch sensor on both front corners. An example of such a robot constructed using the Mindstorms is shown in Figure 1.

[1] Typically a tricycle that has 2 driven wheels, each with its own motor, and the third wheel being a caster

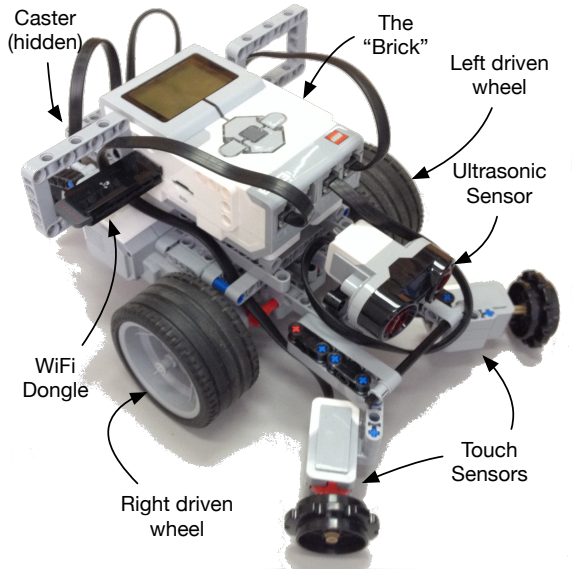


Figure 1. The Lego Mindstorms robot of the explorer behavior example.

The first part of the code, below, takes care of connecting the program to the Mindstorms by reifying the different motors and sensors as variables:

```
(var motA := [LRPEV3Bridge motorA])
(var motB := [LRPEV3Bridge motorD])
(var ultra := [LRPEV3Bridge sensor3])
(var righttouch := [LRPEV3Bridge sensor1])
(var lefttouch := [LRPEV3Bridge sensor4])
```

Five variables are declared and immediately initialized, which is mandatory. In LRP, code between square brackets are actions, *i.e.* Smalltalk blocks. The class `LRPEV3Bridge` is a facade class responsible for connection to the Mindstorms and making the different sensors and motors available. This is in essence how LRP code interacts with specific robot platforms: reifying relevant elements as variables and subsequently interacting with these variables in actions, *i.e.* in Smalltalk code.

With the variables defined, the definition of the state machine for the behavior starts as below. The machine is called Dora (for Dora the Explorer), and initially defines two states and two transitions:

```
(machine Dora
  (state forward
    (onentry
      [ motA value startAtSpeed: 55.
        motB value startAtSpeed: 55.])
    (onexit [motA value stop. motB value stop]))
  (state looking )
  (ontime 600 forward -> looking t-look)
  (ontime 120 looking -> forward t-forward)
```

Lines 7 through 11 specify the `forward` state. The block in lines 9 and 10 is executed whenever this state is entered. As it represents the robot moving forward, both motors are started at 55% of top speed. The block in line 11 is executed whenever the robot leaves the `forward` state and therefore stops both motors. The `looking` state in line 12 does not define any actions.

Note that both blocks use the motor variables defined in lines 1 and 2, and always send them the `value` message first. This is because all variables are in fact Smalltalk `ValueHolders`, as we will discuss in Section 4.2.

Lines 13 and 14 show two timeout transitions. The numbers given in the transitions specify a timeout in milliseconds, starting from when the source state is entered, and trigger after the timeout is reached. The text of the remainder of the transition specifies, respectively source state, destination state, and transition name.

With this code in place, the robot alternates between moving forward for 0.6 seconds, and then waiting for 0.12 seconds. In those 0.12 seconds the three different sensors are polled (which takes a bit less than 0.12 seconds), as is defined in the next three lines of code:

```
15 (event wall [ultra value read < 20])
16 (event rightbump [righttouch value read = 1])
17 (event leftbump [lefttouch value read = 1])
18 (on wall looking -> backup t-backup)
19 (on rightbump  looking -> backup t-rt-backup)
20 (on leftbump looking -> backup t-lt-backup)
```

Lines 15 through 17 define events. The interpreter will evaluate the actions for these events only if triggering these events can cause a transition to occur. In this case, the transitions on line 18 through 20 may occur as they start from the `looking` state and go to a `backup` state, defined below.

In summary: if none of the events trigger, the robot goes to the `forward` state, otherwise it goes to the `backup` state.

```
21 (state backup
22   (onentry
23     [ motA value startAtSpeed: -32.
24       motB value startAtSpeed: -32.])
25   (onexit [motA value stop. motB value stop]))
26 (ontime 300 backup -> turn t-turn)
27 (state turn (onentry
28     [ motorA value startAtSpeed: -32.
29       motorB value startAtSpeed: 32.])
30     (onexit [motA value stop. motB value stop]))
31 (ontime 700 turn -> forward t-tforward)
32 )
33 (spawn Dora forward)
```

The above `backup`, and `turn` states, together with the `t-turn` and `t-tforward` transitions implement the behavior of backing up, turning around, and resuming moving forward. The last line of code specifies that the `Dora` machine should be started by the interpreter and that its initial state is `forward`. This `spawn` statement also can be used as an action in an `onentry` of a state, which means that when this state is entered the specified machine should be interpreted.

This code is sufficient for implementing the explorer behavior. When editing this code in LRP, the interpreter is always running and updating the interpreted machine while the programmer types, and moreover the LRP window, shown in Figure 2, displays the tree of current machines, the contents of variables, and a visualization of the machine. The visualization highlights the currently active state (`looking` in the figure) and the last taken transition. Also, variables can be inspected and their values set.

3. Bridging LRP to Robot Hardware: Controlling the Mindstorms

LRP is at its core a live programming language for nested state machines. It is implemented in Pharo, using PetitParser [10] as the parser generator, Roassal2 [2] for the visualization of the state machines, and Spec [11] to build the user interface.

The language features have been designed with the use as a robotics behavior layer in mind, yet the language itself does not have any intrinsic robotics support. This responsibility instead lies on bridging software that spans the gap to specific robot platforms. Currently, LRP comes with a bridge to ROS [5], and the Lego Mindstorms EV3 [13] via JetStorm [7]. In this section we present the latter and discuss a practical issue we faced programming the Mindstorms.

3.1 Hard- and Software

The Lego Mindstorms EV3 [13] is the third iteration of the Lego Mindstorms line. The embedded system of the set is called the *brick*, and it features an 300 Mhz ARM9-based processor, 64MB of RAM which runs Linux 2.6.x. The sensor package (in the education version) is an ultrasound distance sensor, two touch sensors, a color sensor and a gyroscopic sensor. Three motors are supplied, each motor with a built-in rotation sensor. Last but not least, a comprehensive set of Lego bricks are included, enabling the speedy construction of a wide variety of robot hardware.

The brick also includes an USB port, and support for one specific WiFi dongle, which allows the robot to be remote-controlled via WiFi. JetStorm [7] is a Pharo package that allows for the remote control of the EV3 by reifying the brick, sensors and motors as Smalltalk objects that can be sent messages. For example, sending the `startAtSpeed: 55` message to a motor object causes a command to be sent to the brick to start the corresponding motor at 55% of the top speed the motor is capable of.

The LRP Mindstorms bridge currently consists of a facade class `LRPEV3Bridge` that is placed in front of JetStorm. This class provides features for connecting to the brick over IP and retrieving the various sensors and motors connected to the brick. The latter is shown in lines 1 through 5 of the example program. If there is no IP connection to the brick when a sensor or motor is retrieved, the user is prompted for the IP address of the brick and a connection is set up.

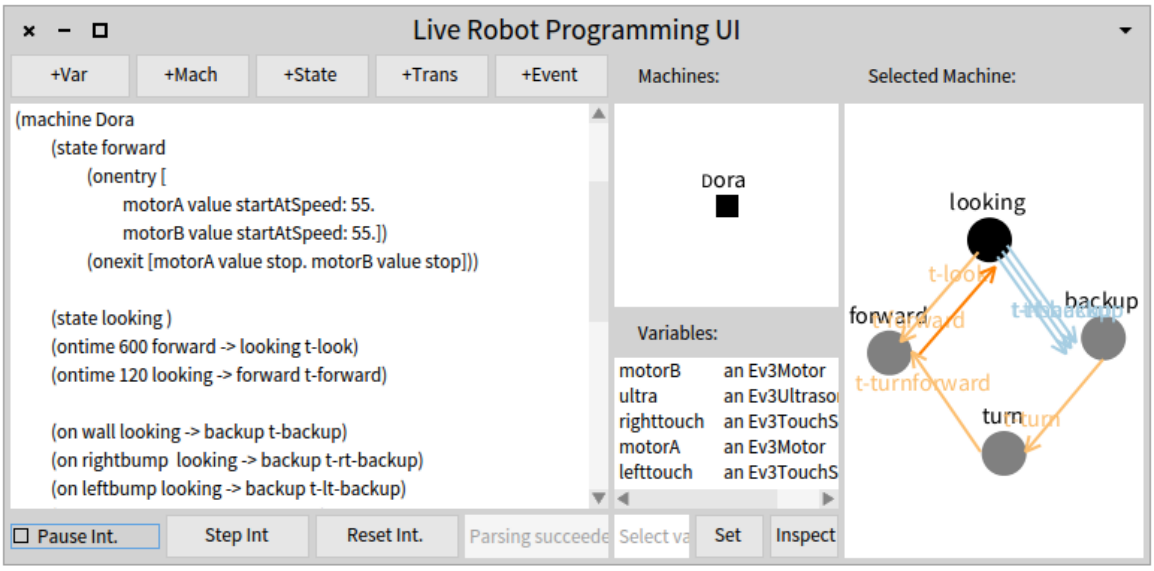


Figure 2. The LRP editor showing part of the example of this text: the Dora machine.

The various sensors and motors that are retrieved are objects provided by JetStorm, no facade is placed in front of them.

In our experience, the one, minimal, facade class has proven to be sufficient to allow small experiments with the Mindstorms. We are however faced with the situation that LRP may grow to have multiple bridges to many different robot API's. For example the API to ROS is quite different. It requires movement vectors to be sent, and their interpretation by the robot eventually causes the respective motors to operate. A wide disparity in how these APIs are exposed to LRP programmers will cause a tight coupling of LRP programs to a specific API and prohibit reuse of behaviors across robot platforms, effectively splintering the language in different versions for different APIs. It would therefore be beneficial to have at least some basic uniformity of the API that the different LRP bridges expose, at least when considering the lowest common denominator of the APIs. Consequently this could possibly require the EV3 Bridge facade to increase in complexity, translating the common API calls to JetStorm calls. We consider the study of such a common API as future work.

3.2 Experience Report: The Issue of Lag

Live Programming of the Lego Mindstorms is a very satisfying experience. It is possible to quickly prototype reasonably complex behaviors, while benefitting from the immediate feedback that live programming brings. We are able to change the behavior of a robot *while it is running* and active in its environment, and the visualization of the state machine allows us to immediately establish in which state the robot is and how it got there. There is only one negative point in the entire experience, and that is the presence of network lag on robot commands.

Sending commands from a computer to the brick over the network and waiting for a reply causes a notable delay in interactions of the LRP interpreter with the robot. Informal microbenchmarks have shown us that it takes approximately 30 microseconds for a sensor read operation to return the sensor's value, and the same time to instruct a motor to start. While this time lag may seem negligible, this turns out not to be the case. For example, in line 15 to 17 of the example code, three sensors are polled, which therefore takes approximately 120 microseconds. This is a noticeable delay, and a time in which the robot may advance a significant distance. For the example the distance traveled in that time is 5 cm, with the motors at 55% of top speed.

It is exactly because of this delay that the Dora behavior is structured in a `moving` and a `looking` phase. The robot first moves for a distance that is deemed 'safe', and then stops to verify if the wall is too close. This results in a stuttering behavior of the robot that is quite noticeable. If reading sensors were immediate, there would be no need for a `looking` state: the robot would continuously poll the sensors for their data. As a result the robot would not stutter and be able to explore at a higher overall speed.

We have experience with programming robots on the predecessor of the EV3, having software run on the brick itself by using the leJOS [6] Java to NXT cross-compiler. While the old brick has significantly inferior hardware, this setup is orders of magnitude more responsive, resulting in robot behaviors that are much more fluid and faster. Consequently, Dora-like behaviors for example can be executed much faster. Note that these experiments were in Java code

and hence did not suffer from any overhead of the LRP interpreter. The overhead of LRP is however almost negligible: in informal tests, the overhead for evaluating events and executing state transitions has been benchmarked to be around one millisecond.

Ideally the robot behavior software would therefore run locally on the EV3 itself. There is however, as yet, no support in Pharo for running on the EV3. As a point of reference, only recently (April 2014) has the first beta release of leJOS on the EV3 been made available. We have not yet been able to experiment with it, nor do we have the resources required to reimplement the LRP interpreter in leJOS.

4. Implementing LRP

The interpreter of LRP is at its core a plain interpreter implementation for nested state machines, extended in two ways for live programming [4]. First it is robust with respect to incomplete programs and keeps on executing in the face of errors. Second it is able to modify the state machine while it is running, adding and removing elements without always requiring a restart.

There are two pieces of the implementation of the interpreter that we discuss here, as they show how the use of Smalltalk has aided us in its implementation, what are limitations due to the implementation and how we plan to address them. These two elements are compilation of the blocks and variables as `ValueHolders`.

4.1 Compiling the Blocks

Actions are used to connect LRP to the API of the specific robot platform. They also may need to perform any kind of computation on sensor inputs and the state of variables to establish whether events occur, and hence may also need to update variables at some point. As a result we found it a natural choice to allow actions to have the full power of Smalltalk available and hence have them be Smalltalk blocks.

Having actions as blocks however raises issues of performance. While the behavioral layer is not a time-critical element in the software of the robot, it does form part of a computation chain that goes from sensor readings up to actuator actions. As such, any overhead that it adds in this process does have effect on the performance of the robot. For this reason, we decided that the overhead of executing actions must be minimal. Hence, in the interpreter actions are compiled blocks: to run them only the `value` message needs to be sent.

Parsing in LRP is performed by PetitParser [10], and action blocks are also parsed, using the Smalltalk parser that is part of PetitParser. As a result, when the interpreter is passed the AST of the state machine to interpret, these blocks have the form of ASTs. The interpreter traverses the complete AST for the program and compiles all action blocks. The result of the compilation of an action block is a `BlockClosure` that has references to all the variables in scope and hence just needs to be sent the `value` message to execute.

The process of compiling the AST of an action block is as follows:

1. a `Dictionary` is created of all variables in scope, taking into account shadowing of variables.
2. Text for the signature for a method is created of the form `captureV:V:V:`, taking as many `V:` arguments as the number of variables in the dictionary.
3. The names of the parameters of this signature are the keys in the dictionary. In the body of the method, the LRP variables are hence in scope of the Smalltalk code.
4. The signature is appended with the string `'^ [1]'` and this complete method definition string is parsed.
5. In the resulting AST method, the subtree for `'[1]'` is swapped with the AST of the block to compile.
6. This method AST is compiled.
7. The resulting method is invoked, passing it the values of the variables in the correct order. This causes the `BlockClosure` to capture variable references such that they may be used inside the code of the action.

For example, let us consider the onentry block of lines 9 and 10 of the Dora example. The result of step 5 is the AST for the following:

```
1 captureV: motB V: ultra V: righttouch
2         V: motA V: lefttouch
3   ^ [motA value startAtSpeed: 55.
4      motB value startAtSpeed: 55 ]
```

Step 6 yields a `CompiledMethod` for the above, *i.e.* a method whose execution returns the `BlockClosure` that corresponds to the action (lines 3 and 4). In step 7, this method is invoked with as arguments the values of the variables `motB`, `ultra`, `righttouch`, `motA`, `lefttouch`. The returned `BlockClosure` has hence captured the references for the variables it uses (`motA` and `motB`). This allows the action to be executed by simply sending the `value` message to this `BlockClosure`.

The compilation of action blocks has turned out to be quite straightforward to implement, taking only about 20 lines of code (of arguably low complexity). We consider that being able to achieve such a complex task so succinctly is a testament to the power and flexibility of Pharo Smalltalk.

There are however two downsides to the current implementation. Firstly, the method that is compiled has no class and an incorrect source code pointer. In our experience this has caused issues when programming: the block cannot be printed, the debugger does not work correctly and in some cases even primitive error handling fails, causing Pharo to crash. An important avenue of future work is to improve the compilation process such that these issues are addressed.

Secondly, methods can only take up to 16 arguments. Consequently, if there are more than 16 arguments in scope, compilation of the action block fails. A possible mitigation of this issue would be to perform a semantic analysis of the block to establish which variables are effectively used inside the block and only pass these as arguments in step 2,3, and 7 above. We also consider this as future work.

In summary We were able to incorporate the full power of an OO language in our state machine-based language thanks to the fact that we have straightforward access to the following:

- a parser of Smalltalk expressions that produces ASTs,
- ASTs of methods allowing for their compilation at runtime, isolated from a class definition,
- blocks that capture the arguments of their enclosing method when they are created.

4.2 LRP Variables are ValueHolders

In LRP, variables are key to interact with specific robot platforms. This is because they are used to reify API elements from these platforms and the code in actions interacts with these elements. For example, in the Dora example above, actions start and stop motors and poll sensors. Variables however serve as more than that, and this can be already seen in the Dora example. The example contains many magic numbers, *e.g.* motor speeds, minimal wall distance (in line 15), and timeouts for the different transitions. All these numbers can (and actually should) be replaced by the use of variables, turning these magic numbers into robot calibration constants. Beyond cleaning up the code, this has as consequence that they can then be modified in the LRP editor while the program runs, effectively recalibrating the robot as it runs. Lastly, if the turning time on line 31 would be a variable, it could contain a random number that is set every time a turn is about to begin. This randomizes the turns, making the exploring behavior immune to being stuck in a loop. Because all of the above reasons, variables must truly be mutable.

Yet these variables are used by three different entities: the original program AST that contains the result of variable initialization, the LRP editor, and the different actions that use these variables. Recall that these blocks get passed these variables by reference when they are constructed, as discussed in Section 4.1. As a consequence, any change to the values of variables is invisible to these blocks! This is because changes to the values do not affect the references that were passed to the blocks as they were constructed. Hence variables may not be changed.

To address the issue that values of variables may not be changed yet at the same time they must be mutable, we have made use of ValueHolders. Every variable is a `ValueHolder` that contains the value. This however entails that reading the value of a variable requires sending the `value` message to the variable, and setting the value of a variable is using the `value:` message instead of normal assignment.

Our experience has shown in practice that in the beginning of writing LRP code it is easily forgotten that variables are `ValueHolders`, leading to widespread errors in behaviors. Such errors are however quickly revealed: simple variable accesses usually already cause problems as the `ValueHolder` class implements few messages. We are planning transparent use of `ValueHolders`, *i.e.* not requiring the use of the `value` and `value:` messages, as future work. We have first considered source code manipulation of the code in the block to automatically transform accesses and modifications to the use of this messages. This however does not address the issue of the variables being used and modified outside of the block, *e.g.* when they are passed as method parameters. A second possible path would be to try the new Slots mechanism. We would have variables as slot instance variables of a purpose-built class. The slot reading and writing mechanism would then implement the extra indirection that is currently achieved by the `ValueHolders`. As the Slots mechanism has not been fully implemented its suitability is however yet to be determined.

In summary We required the use of a double indirection to be able to have mutable variables, and the `ValueHolder` mechanism has shown to be a fitting solution. Requiring the use of `value` and `value:` messages in actions is however suboptimal, and we are planning solutions to this issue.

5. Related Work

Considering robot behaviors using nested state machines, two languages and tools are well-known: The Kouretes Statechart Editor (KSE) [14] and XABSL [8]. In KSE state machines are graphically edited, with an option to start from a text-based description. The tool then follows a model-driven process to generates the executable code for these machines. XABSL is text-based, using an XML representation of the state machines. A variety of support tools are present, for example, a tool that creates (static) diagrams of the machine.

None of the languages above provide any support for live programming, and the live programming languages below do not consider state machines as their computational model.

Live programming was first proposed by Tanimoto [12]. The language presented in that work is VIVA, a visual programming language for image manipulation. More recently, McDirmid proposed the SuperGlue language [9], based on dataflow programming and extended with object-oriented constructs. Live programming of the UI has been proposed by Burckhard *et al.* [3], by adding specific features for live UI construction to an existing programming language. The keynote of Victor [15] shows multiple live programming examples in Javascript, producing pictures, animations and games. A recent addition to live programming is the Swift language by Apple [1], which allows for live programming in specific workspaces called Playgrounds.

6. Conclusion and Future Work

In this paper we have reported on our first experiences of writing Live Robot Programming (LRP) programs for the Mindstorms EV3, and detailed how some of the features of Pharo Smalltalk allowed us to accomplish its implementation.

We first gave a brief overview of LRP through the use of an example program. The program implements a space exploration behavior on a differential drive robot constructed using the Mindstorms (illustrated in Figure 1). We then discussed the LRP bridge to the Mindstorms. LRP allows for the live programming of robot behaviors, yet is not linked to a specific robot platform, instead relying on such bridging software. This was followed by an experience report that focused on how the lag in sending commands to the EV3 negatively impacts robot performance. We then discussed how specific features of Smalltalk have aided in the construction of the LRP interpreter, more specifically the parsing and AST manipulation and compilation support, blocks and ValueHolders.

There are multiple avenues of future work, which we have discussed in some detail along this text. In summary, these avenues consist of the study of a minimal common API for the bridges to different robot platforms, improvements of the compilation process of action blocks, and elimination of the ValueHolder messages for variables.

In our experience, live programming for robot behaviors yields an order of magnitude faster development time, and is a key enabler of fast prototyping of and experimentation with behaviors. Lastly, without the language features of Smalltalk and all the infrastructure available in Pharo its implementation would have been much more demanding, if not impossible, to realize with the resources at our disposal.

More Information, Availability

The home page of LRP is `http://pleiad.cl/LRP` The implementation of the language is open source, MIT license, and download instructions are on its home page.

References

[1] Apple, inc. Introducing swift. https://developer.apple.com/swift/.

[2] Alexandre Bergel, Damien Cassou, Stéphane Ducasse, and Jannik Laval. *Deep Into Pharo*. Square Bracket Associates, 2013.

[3] Sebastian Burckhardt, Manuel Fahndrich, Peli de Halleux, Sean McDirmid, Michal Moskal, Nikolai Tillmann, and Jun Kato. It's alive! continuous feedback in ui programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 95–104, New York, NY, USA, 2013. ACM.

[4] Johan Fabry and Miguel Campusano. Live robot programming. In Ana Bazzan and Karim Pichara, editors, *Advances in Artificial Intelligence - IBERAMIA 2014*, number 8864 in Lecture Notes in Computer Science. Springer Verlag, 2014.

[5] Open Source Robotics Foundation. ROS.org: Powering the world's robots. http://www.ros.org.

[6] The leJOS Group. leJOS: Java for LEGO mindstorms. http://www.lejos.org/.

[7] Jannik Laval. Jetstorm - a communication protocol between Pharo and Lego Mindstorms. Technical Report 140616, Mines-Telecom Institute, Mines Douai, jun 2014.

[8] Martin Lötzsch, Max Risler, and Matthias Jüngel. XABSL - A pragmatic approach to behavior engineering. In *Proceedings of IEEE/RSJ International Conference of Intelligent Robots and Systems (IROS)*, pages 5124–5129, Beijing, China, 2006.

[9] Sean McDirmid. Living it up with a live programming language. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 623–638, New York, NY, USA, 2007. ACM.

[10] Lukas Renggli, Stéphane Ducasse, Tudor Gîrba, and Oscar Nierstrasz. Practical dynamic grammars for dynamic languages. In *4th Workshop on Dynamic Languages and Applications (DYLA 2010)*, Malaga, Spain, June 2010.

[11] Benjamin Van Ryseghem, Stphane Ducasse, and Johan Fabry. Seamless composition and reuse of customizable user interfaces with spec. *Science of Computer Programming*, (0), 2014. In Press.

[12] Steven Tanimoto. VIVA: A visual language for image processing. *Journal of Visual Languages & Computing*, 1(2):127–139, June 1990. http://dx.doi.org/10.1016/S1045-926X(05)80012-6.

[13] The LEGO group. LEGO MINDSTORMS Education EV3. https://education.lego.com/mindstorms.

[14] Angeliki Topalidou-Kyniazopoulou, Nikolaos I. Spanoudakis, and Michail G. Lagoudakis. A case tool for robot behavior development. In Xiaoping Chen, Peter Stone, LuisEnrique Sucar, and Tijn Zant, editors, *RoboCup 2012: Robot Soccer World Cup XVI*, volume 7500 of *Lecture Notes in Computer Science*, pages 225–236. Springer Berlin Heidelberg, 2013.

[15] Bret Victor. Inventing on principle. Invited Talk at CUSEC'12, 2012. Video recording available at http://vimeo.com/36579366.

Modern Problems for the Smalltalk VM

Boris Shingarov
boris@shingarov.com

Abstract

I propose an approach to managing new classes of Smalltalk VM complexity which emerged due to recent advances in technology, through the execution of the VM on formal models of the target processor. Three examples of this approach are discussed. First, I describe out-of-ISA-band observation of the VM based on full-system simulation in which the simulator is aware of the Smalltalk semantics. I also describe experiments in mechanical co-synthesis of the VM and the simulator from the same formal Processor Description Language, leading to an automatically-retargetable JIT. The major obstacle to the usefulness of this approach is the PDL's suitability for toolchain synthesis. Finally, an experimental attempt to bridge from hardware structure directly to JIT is discussed.

Categories and Subject Descriptors D.3.4 [*Programming languages*]: Processors

Keywords Smalltalk Virtual Machine, Full-System Simulation, Processor Description Language, Hardware-Software Codesign, Instruction Set Architecture

1. Introduction

Since the fundamental design of the major Smalltalk VMs has stabilized in the 1990s, the computing landscape has undergone radical evolution. The merge of technologies traditionally characteristic of high-performance computing with embedded computing, the System-on-Chip revolution, the end of the uniprocessor era — all these changes necessitate further evolution of the Smalltalk VM. One way to classify these changes is by position relative to the Smalltalk VM: the application workload we run on Smalltalk, vs. the computing platforms we run Smalltalk on. A shared characteristic of both is an increase, and a change in the nature, of complexity that the VM designer has to deal with.

Application workloads today often expose software defects related to multiprocessor/multicore parallelism, race conditions, or complex compiler optimizations; traditional source-level debugging techniques [Rosenberg'96] are ineffective in these situations and result in prohibitively high debugging effort. In-band observability aids (similar to DTrace) do help factoring the complexities involved in such defects to a limited extent.

The meaning of "performance" has been largely redefined with the end of the uniprocessor era. In traditional VM design, we mostly thought of "processor performance" in terms of sequential instructions-per-second, and of Smalltalk performance in terms of bytecodes-per-second and sends-per-second, n-code cache size-efficiency, and other metrics of similar nature. We now have very fast JITs making maximum use of the processor's ILP features, saturating the hardware in terms of bytecodes-per-second; very efficient PICs giving maximum sends-per-second; yet today's metrics of performance have changed, so we would consider trading sends-per-second for better power efficiency, as MIPJ (Million Instructions Per Joule) has become a more important performance characteristic than MIPS (Million Instructions Per Second). A typical Big Data customer today is concerned with the efficiency of the fundamental design of traditional Smalltalk object memory model, which imposes high FFI marshalling overhead and high d-cache miss rate due to high pointer dereference rate ("pointer-chasing") inherent in such object memory.

Modern architectures on which we run Smalltalk are undergoing a fundamental change which can be characterized as "the end of the uniprocessor era". Dave Patterson argues [Patterson'06] that the main challenges of today's computer architecture are the following "walls":

- Design cost wall
- Software legacy wall
- Power wall
- Memory wall
- ILP wall

These new classes of complexity mean the VM researcher has a new task before them: to create new observation strategies for getting insight into the Smalltalk VM. These mechanisms would allow us to regain "the clearness and ease of comprehension" (to use David Hilbert's expression) of the VM's behavior in today's complex post-uniprocessor computing environment; *"for what is clear and easily comprehended attracts, the complicated repels us"*.

2. Out-of-Band VM Observation

The Instruction Set Architecture (ISA) is the interface between software and the processor; when the program (for example, the Smalltalk VM) is running, this interface can be thought of as a communication channel between the program and the processor (one could imagine the instruction stream to be the program-to-processor direction of that channel, the other direction being the execution results: register values, flags, branching outcome, traps, etc.; we should also not forget about external signals flowing into the processor). Generally the mechanisms that we use to gain an understanding of what is happening in the software, can be classified by their positon relative to that communication channel as either In-Band or Out-of-Band.

Traditional debuggers [Rosenberg'96] are an example of In-Band mechanism. In-band mechanisms are inherently limited in what kind of information they can provide about the system being investigated. They also lack deterministic repeatability as well as reverse execution capabilities. This deficiency renders them incapable of aiding in understanding such important classes of situations as — to point out only one class — race conditions in parts of the system driven by asynchronous external events (e.g. network interrupts). In-band mechanisms are also destructive to the machine state (in other words, they are not truly transparent to the observed system). For example, hitting a hardware breakpoint will cause the execution of the breakpoint interrupt service routine and of the whole chain of context transitions, changing the state of the machine at the system level, and at a minimum, destroying the state of the memory cache hierarchy; in the end, analysis of e.g. timing-related phenomena becomes impossible under this class of debuggers.

In contrast, an Out-of-Band observation mechanism is any observation mechanism which does not work over the same ISA interface to the processor as the observed system. In the rest of this paper, I will talk about the following out-of-band mechanisms and my experiments to use them for better understanding of the Smalltalk VM:

- Functional simulation of the system at different levels of fidelity (instruction-accurate, cycle-accurate, etc.);
- Software simulation of a structural model of hardware;
- Custom-Instrumented FPGA models of hardware.

In section 6, I describe an experiment in which both the VM and (the simulator for) the machine on which the VM executes, are both automatically derived from the same processor description written in a formal language.

3. Full-System Simulation

Not to be confused with *emulation* (which focuses on mimicking the function of the emulated system; Bochs and QEMU are examples of processor emulators), system *simulation* is concerned with modeling the internal state of the simulated system. Simulation can rougly be generally classified into structural and behavioral. An extreme example of a structural model of a microprocessor is Michael Steil's transistor-level simulation of MOS 6502. Verilog simulation of the RTL-level hardware design of OpenSPARC T1 would be a more typical example of structural simulation.

Behavioral simulation is faced with trade-offs between completeness, accuracy, and efficiency. At the low side of the completeness scale are user-program-level simulators. At the opposite high side are full-system simulators. "Full-system" means that the simulation's functional fidelity is complete enough to run the whole operating system and applications unmodified and unaware of running in a simulation. Orthogonal to the measure of completeness is the measure of accuracy, i.e. regard of the model to certain aspects which do not affect the model's ability to execute the operating system quite fine in other aspects. The prime example of this concept is *timing accuracy*, roughly dividing simulators into *instruction-accurate* and *cycle-accurate*. Accuracy comes at the price of efficiency. Using more precise terms, in SystemC TLM language, the lowest level of timing accuracy is *software timing*: the time unit is one instruction, and the timing of memory hierarchy is not modelled at all (nor do these software-timed simulations model other microarchitectural time aspects such as out-of-order execution). Virtutech Simics in native mode and IBM CECsim use this level of time model accuracy.

Loose Timing and Approximate Timing are the next two levels of time modeling. These more accurate models come with progressively higher performance cost. In recent years, there has been a proliferation of full-system simulators offering extremely accurate microarchitechure-level models. Taken together, the Simics Microarchitectural Interface, Flexus, GEMS [Martin'05], Opal (formerly called TFSim [Mauer'02]), M5 [Leupers'10], GEM5 [Binkert'11], present a rich variety of cycle-accurate simulation approaches and provide models for a wide selection of ISAs.

4. Smalltalk-Aware System Simulation

Today's prevalent full-system simulators allow complete programmatic access to all aspects of the simulation. The fundamental event model, the processor, the memory hierarchy, and the peripheral devices are all modelled by loosely-coupled modules interacting via open APIs. These APIs allow automatic detailed analysis of the simulated execution. There are numerous modules available today providing full symbolic debugging of C programs, deep awareness of various operating systems' interal functionality, analysis of execution of network stacks, etc.

Wright et al. [Wright'06] use a similar approach for out-of-band introspection of the HotSpot JVM. Using full-system simulation of the SPARC processor, they were able to gain significant new insight into the interaction of the VM

with the memory hierarchy, and in particular into the effect of GC and n-method recompilation on cache efficiency.

In that light, full-system simulation appeared like a promising approach to gain new understanding of the Smalltalk VM. In one experiment, I created a module which makes the simulator aware of the semantics of execution of the Cog JIT VM. Because at the time of these experiments Cog JIT only ran on IA32, the FSS system selected for this experiment was Simics simulating an in-order Intel x86 processor running a stock "Tango" target (a Fedora Core Linux). The module introspects into the receiver object at any instruction when the processor is executing n-code. This is done using the simulator's Python API (for exploration, it felt more "immediate" than the also-available C API). First, the receiver oop, which in IA32 Cog JIT is kept in the `%EDX` register, is obtained from the state of the simulated processor:

```
oop = conf.cpu0.edx
```

The following simple function then does some reasoning about the object:

```
def print_class_of_oop(oop):
  if ((oop & 1)==1):
    print "SmallInteger"
  else:
    headerType = smalltalk_headerType(oop)
    if (headerType==3):
      print "...looks like compact class..."
    else:
      word2 = read_virt_value(oop-4, 4)
      classOop = word2&0xFFFFFFFC
      print "class oop: ", hex(classOop)
      clsNameOop = read_virt_value(classOop+32, 4)
      print "class name oop: ", hex(clsNameOop)
      str=""
      for offset in range(smalltalk_objByteSize(clsNameOop)):
        str += "%c" %
             read_virt_value(clsNameOop + 4 + offset, 1)
      print str
```

First we look at the tag bit to see if the oop is a pointer or a SmallInteger. A SmallInteger does not leave much to further introspection. With a pointer, we dereference it to access the object header:

```
def smalltalk_headerType(oop):
  return read_virt_value(oop, 4) & 3
```

where memory is accessed like this:

```
# Read a little-endian value from a physical address.
def read_phys_value(paddr, len):
  cpu = conf.cpu0
  val = 0L
  for i in range(len-1, -1, -1):
    try:
        val = (val<8) | SIM_read_phys_memory(cpu, paddr+i, 1)
    except:
        raise PTrackError("%s can not read byte at p:0x%x"
                          % (cpu.name, paddr+i))
  return val

# Read a little-endian value from a virtual address.
def read_virt_value(vaddr, len):
  cpu = conf.cpu0
  try:
    paddr = SIM_logical_to_physical(cpu, Sim_DI_Data, vaddr)
  except:
    raise PTrackError("%s can not translate v:0x%x to physical"
                      % (cpu.name, vaddr))
  return read_phys_value(paddr, len)
```

(The simulator interprets the state of the simulated MMU and performs address translation, modeling the details of the processor's TLB in `SIM_logical_to_physical()`). The crucial difference between this simulator-side read and printing a memory value in a traditional debugger is that the simulator-side read (and in fact, none of the operations in this module) does not disturb the state of the processor. If the simulation includes some sort of cycle-accurate model of time (e.g. a model of the memory hierarchy), oop dereferencing will not affect the state of that model. This is in contrast with the situation where the JIT dereferences the oop: such debugee-side access will cause simulated cache misses, pipeline stalls, or whatever other possible effects of the "`load`" instruction are modelled at the present accuracy level (cf. next section).

The simplest way to try the introspection module is to stop the simulator at a "magic breakpoint" in the middle of n-code. We insert a no-effect instruction into the emitted n-code. This "magic instruction" should not only have no effect but also be extremely unlikely to occur in real code. For example, on IA32 a usual candidate is

```
xchg %bx, %bx   # cf. 16r66 below
```

(To make the Cog JIT emit the magic instruction, first an abstract RTL instruction and its IA32 concretization are defined. The abstract instruction is added at the end of the list in `CogRTLOpcodes>>initialize`, and `#initialize` is resent. We also need to add a new method:

```
Cogit>>Magic
  <inline:true>
  <returnTypeC:#'AbstractInstruction*'>
  ^self gen: Magic
```

The IA32 concretization amounts to changing

```
CogIA32Compiler>>dispatchConcretize
  opcode caseOf: {
    ...
    [Magic] -> [^self concretizeMagic].
  }
```

and specifying the instruction encoding by adding this new method:

```
CogIA32Compiler>>concretizeMagic
   "Will get inlined into concretizeAt: switch."
   <inline: true>
   machineCode
      at: 0 put: 16r66;
      at: 1 put: 16r87;
      at: 2 put: 16rdb.
   ^machineCodeSize := 3
```

Once the new instruction is defined, we can modify the JIT to emit it. For illustration purposes, I modified

```
genGetClassFormatOfNonInt: instReg
                into: destReg
                scratchReg: scratchReg
  "Fetch the instance's class format into destReg,
   assuming the object is non-int."
  | jumpCompact jumpGotClass |
  <var: #jumpCompact type: #'AbstractInstruction *'>
  <var: #jumpGotClass type: #'AbstractInstruction *'>
  cogit Magic.  "THIS WILL STOP SIMULATION"
```

```
"Get header word in destReg"
cogit MoveMw: 0 r: instReg R: destReg.
... "rest of method"
```

The VM is regenerated and recompiled after these changes.)

After magic instruction support is installed in the JIT, the simulator is instructed to run the simulation until it encounters the magic instruction; with the simulator paused, our `print_smalltalk_receiver()` function can be invoked from the simulator's command-line interface.

In a more practical scenario, various routines in the Smalltalk-aware module can be driven by callbacks from the simulator. One useful application is collecting statistics. If the simulation includes modeling of microarchitectural details, this technique can provide information about relationships between high-level Smalltalk abstractions and the microarchitectural phenomena: for example, it would be possible to make queries such as "cache hit ratio when performing linked sends, but only when the receiver `isKindOf:` this specified class".

Even more interesting is stopping simulation and returning control to the simulator interface when certain programmatically specified conditions are met. The condition can be as simple as segmentation fault. Quite often in today's parallel computing environments, a bug could be caused by an extremely rare race condition in an asynchronous interrupt-triggered routine, and not reproducible in staged forward execution. This class of problems is extremely well-suited to debugging in simulation. We record some number of last simulation steps in a ring buffer. We run the simulation until the VM crashes. At that point, we can analyze the execution steps that led to the crash (performing the Smalltalk-aware introspection available to us at any step).

5. Deriving a JIT from Processor Description Language

A number of Smalltalk VM implementations have attempted various degrees of retargetability. While the original "PS" VM [Deutsch'83] strictly targeted Motorola 68000, its successor HPS [Miranda:Contexts], [Miranda:Thread] uses the C macro-processor to achieve a degree of portability by providing a "processor definition file" containing an agreed-upon set of "`#define`" C macro definitions. The Cog VM [Miranda:Cog], written in a subset of Smalltalk, contains abstract classes which represent a common concept of what a "processor" can be ("abstract RTL"); these are subclassed to describe concrete target ISAs (such as IA32 or a particular ARM variant).

Being embedded in the VM implementation language and relatively low-level, such ad-hoc processor description facilities require the human implementor of the port to comprehend the details of the processor/platform specifications and manually program the mapping to the "common machine", working in that implementation language. One limiting factor of this approach is the space of processors which can be parametrized given a particular "common machine": often, the set of considerations relevant to a new processor architecture can not be expressed in terms of the existing "common model". Thus, there is a trade-off between specificity (because we need to be able to derive a translator from the processor description) and generality (to cover a wide enough family of processors).

An even more important factor is the complexity of today's architectures and OS platforms. As an example, the ARM Architecture Reference Manual is 5158 pages (as of the ARMv8 "A.a" issue) of natural English language. Moreover, the information in the ARMARM alone is not enough: a VM port would need to take into account considerations of performance optimization, details of the ABI on the platform, etc. The effort on the part of the author of the port to comprehend all this complexity, makes the cost per processor port prohibitive even in the case of universal processor ISAs. The port author is trapped in a cycle of re-interpretation of the natural language of the specifications, as details of his interpretation differ from the processor designer's (ISA implementor's). This kind of porting effort becomes completely unrealistic in today's chip-level-integration systems-on-chip often involving application-specific instruction sets, which are quickly growing to become the norm in embedded applications.

Processor Description Languages [Mishra'08] have been used for automatic synthesis of compilers from formal specifications of processors in parallel with synthesis of hardware, thus eliminating or reducing the problem of software/hardware ISA interpretation discrepancy mentioned above. Pioneering work in PDLs in the 1970s was Gerhard Zimmermann and Peter Marwedel's MIMOLA (Machine Independent Microprogramming Language). Since then, a number of processor description languages and technologies have been developed, such as nML (the processor modeling language in Synopsys' "IP Designer" system), LISA from RWTH Aachen University, EXPRESSION, ArchC, and many others. I chose the open-source ArchC language and system [Azevedo'05], [Mishra'08] as the base for my experiments with deriving a Smalltalk VM from a PDL.

The ArchC system takes a processor description (written in the ArchC language) and generates a simulator for the processor. In addition, the ACCGen compiler generator [Auler'12] takes an ArchC processor description and generates an LLVM "llc" backend.

My experiments with deriving a JIT from ArchC PDL did not involve the LLVM infrastructure. Also, in the proof-of-concept I did not go beyond a trivial mockup of the Cattell algorithm [Cattell'80]; in other words, no optimization was attempted as my focus was on the automatic/unsupervised retargetability and the guarantees it gives against the hardware/software ISA interpretation discrepancies mentioned above.

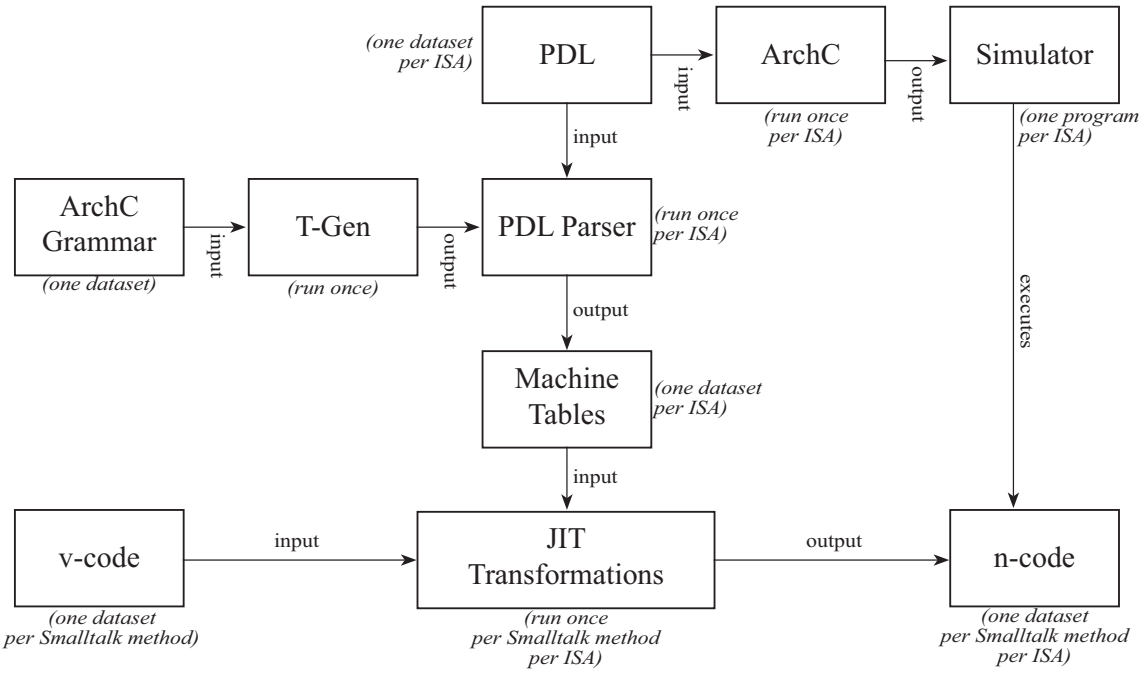


Figure 1. Deriving a JIT from PDL

Figure 1 shows the basic structure of the JIT generator. The same PDL specification of the processor is input to both the ArchC system and the JIT generator. ArchC generates a system simulator allowing out-of-band debugging of the JIT, in line with ideas described in sections 3–5 of this paper. The JIT generator is implemented as a Smalltalk program. With the aid of the T-Gen parser generator, it re-uses ArchC's and ACCGen's PDL grammars. The parser produced by T-Gen, transforms the processor description into Machine Tables which parametrize the Cattell algorithm [Cattell'80]. The Machine Table is a set of productions, and Cattell's algorithm searches for a suitable chain of rewrites leading from the IR goal on the left ("TCOL" in Cattell's paper) to instruction terminals on the right. The experimental Smalltalk JIT is much simpler. While the general Cattell algorithm allows arbitrary and possibly recursive rewrites, my JIT's derivations always consist of two fixed productions: (1) *bytecode ::= RtlInstruction**, and (2) *RtlInstruction ::= MachineInstruction**. The search degenerates into a lookup in the table. In this sense, the JIT is equivalent to any well-known JIT such as Cog or HPS: the difference is not in what code generation it does, but in how it is parametrized automatically from a formal specification rather than manually by the VM programmer. However, with the concept of a Smalltalk JIT derived from PDL now proven possible, there is no reason, in principle, why this approach would not work with more advanced forms of code optimization.

Achieving the derived JIT met with two primary difficulties: (1) the presence of algorithmic definitions in the ArchC PDL, and (2) formal specification of platform ABI.

Extraction of instruction semantics from the ArchC processor description. ArchC's primary focus is on synthesis of efficient simulators. To facilitate this goal, ArchC allows algorithmic expression of instruction semantics. Auler et al. [Auler'12] give the following example of how an instruction behavior may be specified in custom C++ within the PDL model:

```
void ac_behavior( add )
{
  RB[rd] = RB[rs] + RB[rt];
};
```

This approach of being able to specify processor behavior as an algorithm expressed in a universal programming language, favors the point of view of simulation over that of both hardware synthesis and compiler generation. Several authors, including Marwedel and Leupers [Marwedel'94], proposed various approaches to the problem of instruction semantics extraction for the purpose of instruction selection. These approaches generally do not work for ArchC. After all, due to the halting problem it is not even possible to decide, given such algorithmic description of an instruction, whether the instruction's execution will terminate at all, let alone to compute the information necessary for instruction selection. To solve this problem for their C complier, Auler

et al. [Auler'12] propose an RTL-based extension to ArchC. Existing ArchC processor models need to be amended to include this instruction-semantics information. Their ACCGen compiler demonstrates that such amendment does in fact allow to synthesize a working C compiler. My Smalltalk JIT reuses Auler's extensions to the PDL language and the extended models of ARM, SPARCv8, PowerPC and MIPS which are provided with the ArchC distribution.

The major drawback of this approach is that there are essentially two processor definitions (declarative and procedural) and we are again facing the same problem of diverging ISA interpretations.

FFI Glue is the custom machine code that connects the abstract computation machinery of the JIT to the rest of the system which is written in C. Its crucial role is that it's the way for computation to produce effect. The main issue the glue has to deal with, is the platform ABI convention about the C stack which the glue has to synthesize and manage. Most of the time, PDL models don't include an ABI specification. After all, in a full-system simulation such ABI model is altogether not needed because ABI is a software-level concern: it exists entirely within the software system running *on top* of the simulator.

ACCGen partially addresses this problem by extending ArchC to describe calling conventions; however, only their ARM model contains such ABI descriptions.

In light of all this, a production PDL-derived JIT for Smalltalk may need a different substrate for translator synthesis. Ideally, the same processor description would be used to synthesize the hardware, the compiler toolchain (Smalltalk VM included) and the simulator. Machine-readable ABI specs do not need to be integral part of the processor description, but will have to be the accepted starting point of the synthesis of the C toolchain, lest calling convention interpretation discrepancies lead to ABI violations analogous to the hardware-software discrepancy bugs discussed above.

6. Introducing Smalltalk Awareness into Structural Models

In the ideal scenario, the final destiny of a VM synthesized from a high-level processor description, would be running on hardware structures synthesized from the same description (after being debugged in simulation derived from the same description). At the time of this writing, I am not aware of a PDL framework openly available and suitable for such end-to-end experiments with Smalltalk. There is no doubt it will become available in the future. How can we attempt to bridge processor structure and the VM before we have such an end-to-end framework in our hands?

A number of processor instrumentation approaches has been described in the literature [Stollon'11]. EJTAG is a processor debugging facility very specific to MIPS. ARM has a comparable instrumentation system called ETM. I am not aware of open implementations of either EJTAG or ETM,

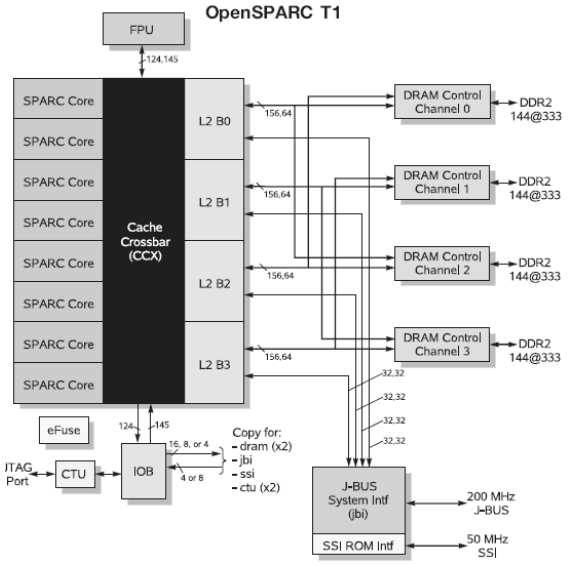


Figure 2. Block Diagram of OpenSPARC T1 [Weaver'08]

and it is not clear how either can be used as the foundation for a research effort to implement Smalltalk-aware processor instrumentation.

Therefore, in one experiment, I attempted to use a hardware description of a complete processor as the starting point. In recent years, open-source processor and system-on-chip IP has matured to a point of significant practical importance in critical production systems. The experiment described here starts from the Verilog source code for the OpenSPARC T1 microprocessor available under the GNU General Public License. It might be possible, using approaches discussed by Marwedel and Leupers [Marwedel'94], to extract enough ISA semantics from this structural description to be able to synthesize a Smalltalk VM, thus closing the hardware-software circle. I have not attempted this yet. Instead, in this experiment, the hardware structure is extended with probes which afford access to the internal state of the hardware at points of interest to the VM researcher and automatic analysis of that state in accord with the programmatic structure of the Smalltalk VM.

The OpenSPARC microprocessor consists of a designer-variable number of processor cores, connected to the memory cache, FPU and other components by the Cache Crossbar (CCX). Figure 2, reproduced from [Weaver'08], shows a block diagram of OpenSPARC T1. In a reference implementation of OpenSPARC on FPGA [Thatcher'08], only the processor core is synthesized from Verilog to the FPGA's programmable logic. The CCX, as well as everything else on the other side of it across from the core (memory interface, FPU, etc., and also some system components which would be off-chip in an ASIC implementation — e.g. Ethernet MAC; I will call the total of these components "the off-core"), are

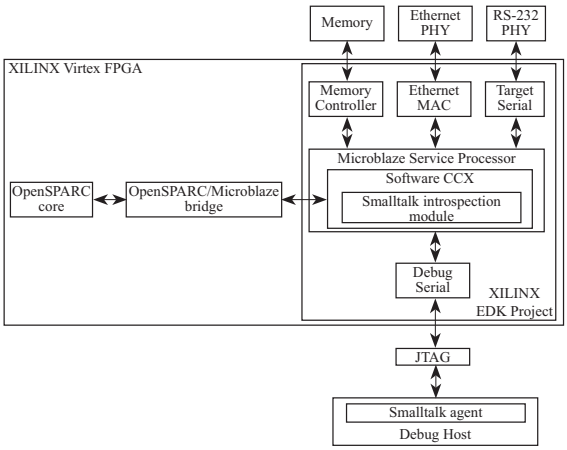


Figure 3. OpenSPARC Implementation on XILINX Virtex FPGA with Smalltalk-Aware Crossbar. Cf. the corresponding diagram in [Thatcher'08], NB the position of the Smalltalk module.

implemented in a XILINX EDK design. Although ultimately both the OpenSPARC core and the EDK project are physically sharing the same programmable logic fabric, the Microblaze service processor, the Ethernet MAC, the memory controller etc. are just standard XILINX IP, and the "off-core" functions are programmed in software running on the Microblaze. This is what facilitates the Smalltalk-aware probe. The off-core code (written in C) is amended with a Smalltalk observation module. The module is controlled by an agent program running on the debug host via the standard Microblaze debug serial port. The module (and the agent through it) have full access to the state of the OpenSPARC, and can reason about Smalltalk objects and VM program-structures similarly to the Smalltalk-aware FSS discussed in Section 4.

Experimenting with this configuration suggests even more importance of deriving hardware and software from a common higher-level processor description (as in section 5 above). Indeed, even the reference implementation of this mainstream microprocessor contains two embodiments (ASIC- and FPGA-oriented) which are not derivative of each other, and hence potentially diverging interpretations of the SPARCv9 ISA.

7. Conclusion

Out-of-band observation techniques can provide deeper insight into crucial aspects of the Smalltalk VM than possible with traditional observation approaches. In particular, this strategy can shed new light on the VM's use of time and power. A JIT can be written in a processor-agnostic manner. Targeting such JIT at a new ISA is a matter of automatically processing the new architecture's PDL description. The same PDL description can also be used for the synthesis of the actual processor, as well as of the simulators for the out-of-band observation, thus eliminating the hardware/software mismatch as a major source of software defects.

References

[Auler'12] R. Auler, P. C. Centoducatte, E. Borin. ACCGen: An Automatic ArchC Compiler Generator. 24th International Symposium on Computer Architecture and High Performance Computing, New York, USA, 2012.

[Azevedo'05] R. Azevedo, et al. The ArchC Architecture Description Language. International Journal of Parallel Computing, 33(5): 453–484, October 2005.

[Binkert'11] N. Binkert, et al. The GEM5 simulator. ACM SIGARCH Computer Architecture News, Vol.39(2), May 2011.

[Cattell'80] R. G. Cattell. Automatic Derivation of Code Generators from Machine Descriptions. ACM TOPLAS, Vol. 2, Issue 2, April 1980.

[Deutsch'83] L. P. Deutsch, A. M. Schiffman. Efficient Implementation of the Smalltalk-80 System. ACM, 1983.

[Hohenauer'10] M. Hohenauer, R. Leupers. C Compilers for ASIPs. Springer, 2010.

[Leupers'10] R. Leupers, O. Temam (eds.) Processor and System-on-Chip Simulation. Springer, 2010.

[Magnusson'95] P. Magnusson, B. Werner. Efficient Memory Simulation in Simics. Proceedings of the 28th Annual Simulation Symposium. IEEE, Phoenix, AZ, USA, April 1995.

[Marwedel'94] P. Marwedel, R. Leupers. Instruction Set Extraction from Programmable Structures. European Design Automation Conference, Grenoble, France, 1994.

[Martin'05] M. M. K. Martin, et al. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. Computer Architecture News, September 2005.

[Mauer'02] C. J. Mauer, M. D. Hill, D. A. Wood. Full-System Timing-First Simulation. Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems. 2002.

[Miranda:Cog] E. Miranda. The Cog Blog. *http://www.mirandabanda.org/cogblog*

[Miranda:Thread] E. Miranda. VisualWorks Threaded Interconnect. Smalltalk Solutions, New York, USA, 1999.

[Miranda:Contexts] E. Miranda. Context Management in VisualWorks 5i. OOPSLA, Denver, Colorado, USA, 1999.

[Mishra'08] P. Mishra, N.Dutt (eds.) Processor Description Languages. Applications and Methodologies. Morgan Kaufmann Publishers, 2008.

[Patterson'06] D. Patterson. Future of Computer Architecture. Berkeley EECS Annual Research Symposium, 2006, Berkeley, Calif., USA

[Rosenberg'96] J. B. Rosenberg. How Debuggers Work: Algorithms, Data Structures, and Architecture. Wiley Computer Publishing, 1996.

[Stollon'11] N. Stollon. On-Chip Instrumentation. Springer, 2011.

[Thatcher'08] T. Thatcher, P. Hartke. OpenSPARC T1 on Xilinx FPGAs. RAMP Retreat, Stanford, August 2008.

[Weaver'08] G. L. Weaver (ed.) OpenSPARC Internals — OpenSPARC T1/T2 CMT Throughput Computing. Sun Microsystems, Calif., USA, 2008.

[Wright'06] G. Wright et al. Introspection of a Java Virtual Machine under Simulation. SMLI TR-2006-159, Sun Microsystems, Calif., USA, 2006.

Reducing Waste in Expandable Collections: The Pharo Case

Alexandre Bergel, Alejandro Infante, Juan Pablo Sandoval Alcocer
Pleiad Lab, DCC, University of Chile

Abstract

Expandable collections are collections whose size may vary as elements are added and removed. Hash maps and ordered collections are popular expandable collections. In the Pharo programming language, expandable collection classes offer an easy-to-use API, however this apparent simplicity is accompanied by a significant amount of wasted resource.

We describe some improvements of the collection library to reduce the amount of waste associated with collection expansions. We have designed a new collection library for Pharo that exhibits better resource management than the standard library. Across a basket of 17 applications, our optimized collection library significantly reduces the memory footprint of the collections: (i) the amount of intermediary internal array storage by 73%, (ii) the number of allocated bytes by 67% and (iii) the number of unused bytes by 72%. This reduction of memory is accompanied with a speedup of about 3% for most of our benchmarks. We further discuss the applicability of our findings to other languages, including Java, C#, Scala, and Ruby.

1. Introduction

Creating and manipulating any arbitrary group of values is largely supported by today's programming languages and runtimes [1]. A programming environment typically offers a collection library that supports a large range of variations in the way collections of values are handled and manipulated.

Collections exhibits a wide range of features [1–3], including being expandable or not. An expandable collection is a collection whose size may vary as elements are added and removed. Expandable collections are typically implemented by wrapping a fixed-sized array. An operation on the collection is then translated into primitive operations on the array, such as copying the array, replacing the array with a larger one, inserting or removing a value at a given index.

Unfortunately, the simplicity of using expandable collections is counter-balanced by resource consumption when not adequately employed [4–6]. Pharo[1] is a dynamically typed programming language which offers a large and rich collection library. Consider the case of a simple ordered collection (`OrderedCollection` in Pharo and `ArrayList` in Java). Using the default constructor, the collection is created empty with an initial capacity of 10 elements. The 11th element added to it triggers an expansion of the collection by doubling its capacity. We have empirically determined that in Pharo a large portion of collections created by applications are empty. As a consequence, their internal arrays are simply unused. Moreover, only a portion of the internal array is used. After adding 11 elements to an ordered collection, 9 of the 20 slot arrays are left unused. Situations such as this one scale up as soon as millions of collections are involved in a computation.

This paper is about measuring wasted resources in Pharo (memory and execution time) due to expandable collections. Improvements are then deduced and we measure their impact.

Research questions we are pursuing are:

A - *How to characterize the use of expandable collections in Pharo?* Understanding how expandable collections are used is highly important in identifying whether or not some resources are wasted. And if this is case, how such waste occurs.

B - *Can the overhead associated with expandable collections in Pharo be measured?* Assuming the characterization of collection expansions revealed some waste of resources, measuring such waste is essential to properly benchmark improvements that are carried out either on the application or the collection library.

C - *Can the overhead associated with expandable collections in Pharo be reduced?* Assuming that a benchmark to measure resource waste has been established, this question focuses on whether the resource waste accompanying

[1] `http://www.pharo-project.org`

the use of a collection library can be reduced without disrupting programmer habits.

Our results shows the Pharo collection library can be significantly improved by considering lazy array creation and recycling those arrays. The expandable collections of Java, Scala, Ruby and C# are very similar to those of Pharo. We therefore expect our recommendations to be beneficial in these languages.

This paper is structured as follows: Section 2 describes the Pharo expandable collections and synthesizes their implementation. Section 3 describes a benchmark composed of 17 Pharo applications and a list of metrics. Section 4 details the use of expandable collections in Pharo, both from a static and dynamic point of view. Section 5 details the impact on our benchmark to have lazy array creation. Section 6 presents a technique to recycle arrays among different collections. Section 7 describes an approach to find missing collection initialization. Section 8 discusses the case of other languages. Section 9 presents the work related to this paper. Section 10 concludes and presents our future work.

2. Pharo's Expandable Collections

The collection library is a complex piece of code that exhibits different complex aspects [7]. One of these aspects is whether a collection created at runtime may be resized during the life time of the collection. We qualify a collection with a variable size as "expandable". An expandable collection is typically created empty, to be filled with elements later on. Typical expandable collections include dictionaries (usually implemented with a hash table), lists, growable arrays in which elements may be added and removed during program execution. Interestingly, expandable collections are designed to only expand. Removing elements from a collection does not trigger any shrinkage of the internal collection. We therefore only focus on element addition and not removal.

2.1 Issues with expansions

Expandable collections are remarkable pieces of software: most expandable collections have a complex semantic hidden behind a simple-to-use interface. Consider the class `Dictionary`. The class employs sophisticated hashing tables to balance efficiency and resource consumption. Such complexity is hidden behind what may appear as trivial operations The programmer has to simply address what to add or remove from the collection while the collection implementation takes care of growing or shrinking the collection accordingly.

Expandable collections commonly used in Pharo employ a fixed-sized array as an internal data structure for storage. Adding or removing elements from an expandable collection are translated into low-level operations on the internal storage, typically copying, setting or emptying a particular part of the array storage.

The creation of an expandable collection may be parametrized with an initial *capacity*. This capacity represents the initial size of the array's internal storage. The size of the collection corresponds to the number of elements actually stored in the collection. Adding elements to a collection increases its size and removing elements shrinks it. When the size of the expandable collection reaches its capacity or close to it, the capacity of the collection is increased, leading to an expansion of the collection. A collection-specific threshold ratio *size / capacity* drives the collection expansion. A 0.75 and 1.0 are commonly used thresholds (0.75 for collections operating with hashtags values and 1.0 for every other collections). Consider the class `OrderedCollection`, a frequently used expandable collection. Consider an ordered collection of a given capacity c. Adding one element to the collection increases its size s by one. When $s = c$, then the collection is expanded to have a capacity of $2c$ elements.

Expanding a collection is a three-step operation summarized as follows:

1. *Creation of a larger new array* – the size of the collection having reached its capacity (*i.e.,* the size of the internal data storage), a new array is created, typically twice as large as the original array.
2. *Copying the old array into the new one* – content of the old array is entirely copied into the first half of the new array.
3. *Using the new array as the collection's storage* – the expandable collection takes the new array as its internal storage, realized by simply making the storage variable point to the new array. The old array is garbage collected since it is not useful anymore.

Although efficient in many situations, expandable collections may result in wasted resources, as described below.

Expansion overhead. Expanding a collection involves creating and copying of possibly large internal array storage. Consider the following micro benchmark:

```
c := OrderedCollection new.
[ 30000000 timesRepeat: [ c add: 42 ] ] timeToRun
  => 3375 milliseconds
```

This benchmark simply measures the time taken to add 30 million elements to an ordered collection. In our current execution setting, the micro benchmarks reported in this section have a variation of 7%.

The class `OrderedCollection`, when instantiated using the default constructor, as above, uses an initial capacity of 10 elements. An expansion of the collection occurs when adding the 11-th element. The capacity is then doubled. The size of the collection is 11 and its capacity is 20. When the 21st element is added to it, its capacity is 40.

Adding 30 million elements in a collection triggers $log_2(30\ 000\ 000 / 10) = 22$ expansions. Such expansions have heavy cost, both in terms of memory and CPU time. When the capacity is equal to or greater than the number of elements to be added:

```
c := OrderedCollection new: 30000000.
[ 30000000 timesRepeat: [ c add: 42 ] ] timeToRun  =>
  => 1356 milliseconds
```

In such a case, no expansion occurs, thus resulting in adding the elements without any expansion phases.

Copying of memory. At each expansion of the collection, the whole internal array content has to be copied into the newly created array. Consider the `OrderedCollection` in which 30 M elements are added to it. Since the collection is expanded 22 times, the internal array has been copied 21 times.

At the first expansion, when the internal storage grows from 10 to 20 slots, 10 slots are copied. Since each array slot is 4 bytes long, 40 bytes have been copied. 80 bytes are copied for the second expansion. Since the internal array size increases exponentially, the number of bytes that are copied scale up easily. Adding 30M elements produces 22 expansions, incurring $\sum_{i=0}^{21} 10 * 2^i = 41$M slot copies. In total, $41 * 4 = 164$Mb of memory are copied between unnecessary arrays. Such copying could be reduced or avoided by giving a proper initial capacity to the collection.

Virtual memory. The memory of a virtual machine is divided into generations. Garbage collection happens by copying part of a generation into a clean generation. Such copying is likely to happen across memory pages [9], since the new array is likely to be in the young generation (*i.e.,* part of the memory used for short lived objects and new object creations). In addition, the copying of arrays may activate part of the virtual memory stored on disk if the part of the memory containing the old array has been swapped to disk [9].

Collector pauses. Garbage collection copies and joins portions of memory to reduce memory fragmentation [10]. Copying and scanning a large portion of memory, such as collections, may cause large and unpredictable collection pause times. The garbage collection pauses in proportion to array size [11].

Unnecessary slots. Expanding a collection doubles the size of the internal array representation. As a consequence, a collection having a size less than its capacity has unused slots.

For example, adding 30 million elements to a collection with the default initial capacity generates 22 expansions. After the 22nd expansion, the collection has a capacity of $10*2^{22} = 41{,}943{,}040$, large enough to contain the 30,000,000 elements. As a consequence, the collection has 41,943,040 − 30,000,000 = 11,943,040 unused slots. Since each slot weighs 4 bytes, nearly 48Mb of memory are unused after having added the 30M elements.

Note that the issue of having unused portion of the array has already been mentioned (Pattern 1, 3, 4 in [12]). Our paper reports the evolution of the amount of unused memory space against the improvement we have designed of the collection library. Our approach to address this issue is new and has not been considered before.

3. Benchmarking and Metrics

To move away from micro-benchmarks and understand this phenomenon better on real applications, we pick a representative set of Pharo applications and profile their execution.

3.1 Benchmark

Appendix A lists the 17 Pharo applications we consider in our benchmark. These applications are open source[2], thus easing a replication of our experiments. These applications are daily used both in industries and academia. They are furthermore supported by active communities.

We employ the benchmark to approximate how expandable collections are used in general.

3.2 Metrics about the collection library

We propose a set of metrics to understand how expandable collections are used and what the amount is of resulting wasted resources. The metrics that we propose to characterize the use of expandable collections for a particular software execution are:

- *NC* – **N**umber of expandable **C**ollections – This metric corresponds to the number of expandable collections created during an execution. This metric is used to give relative numbers (*i.e.,* percentages) for most of the metrics described below.
- *NNEC* – **N**umber of **N**on **E**mpty **C**ollections – Number of expandable collections that are not empty, even temporarily, during the execution.
- *NEC* – **N**umber of **E**mpty **C**ollections – Number of expandable collections to which no elements have been added during the execution. A collection for which elements have been added then removed are not counted by *NEC*.
- *NCE* – **N**umber of **C**ollection **E**xpansions – Number of collection expansions happening during the program execution.
- *NCB* – **N**umber of **C**opied **B**ytes due to expansions – Amount of memory space, in bytes, copied during the expansions of expandable collections.
- *NAC* – **N**umber of internal **A**rray **C**reations – Number of array objects created used as internal storage during the execution.
- *NOSM* – **N**umber of collections that are filled **O**nly in the **S**ame **M**ethods that have created the collections.
- *NSM* – **N**umber of collections filled in the **S**ame **M**ethods that have created them.
- *NAB* – **N**umber of **A**llocated **B**ytes – Accumulated size of all the internal arrays created by a collection.

[2] `http://smalltalkhub.com`

- *NUB* – **N**umber of **U**nused **B**ytes – Size of the unused portion of the internal array storage. For a given collection, this metric corresponds to the difference $capacity - size$.

3.3 Computing the metrics

Measuring these metrics involves a dynamic analysis to obtain an execution blueprint for each collection. We have instrumented the set of expandable collections in Pharo to measure these metrics.

We measure only the collections that are directly created by an application. Computation carried out by the runtime is not counted. If we equally counted collections created by the runtime and the application, a residual amount would have to be determined since collections may be counted several times across different applications.

Collections are often converted thanks to some utility methods. For example, an ordered collection may be converted as a set by sending the message `asSet` to it. Converting an expandable collection into another expandable collection sums up in our measurements.

Our measurements, used to characterize the use of expanded collections and measure wasted resources associated with them, have to be based on representative application executions, close to what programmers are experiencing. Unfortunately, Pharo does not offer a standard benchmark for measuring performance in the same spirit as DaCaPo [13]. We have designed our benchmark from two different sets of program executions: (i) execution of unit tests and (ii) performance scenarios.

Unit-test benchmarks. Running unit tests is convenient in our setting since unit tests are likely to represent common usage and execution scenarios [14]. We execute the unit tests associated with each of the 17 applications.

The primarily purpose of a unit test is to validate correctness. A unit test typically represents a short execution scenario that is quick to run. We will profile the execution of unit tests to measure the creation of short-lived small collections. Executing unit tests is an action often performed by a programmer. Optimizing this action is therefore a valuable contribution. Note that we consider unit tests as part of the applications. This means that collections created within a unit test are counted in our measurement.

Performance benchmarks. We use 15 benchmarks that perform a computation on a large amount of input data. From the 17 applications, we consider 5 applications for which it makes actual sense to run a long execution. These applications are marked with an * and we have three benchmarks for each of these. These benchmarks have been written by the authors of the considered application and represent a typical heavy usage of the application.

Referring to the benchmarks. The tables given at the end of the papers show the result of our measurements. We refer to the execution of unit tests for application X as "Benchmark X", X ranging from 1 to 17. The long executions are referred to as "Benchmark bAST*Y*, bN*Y*, bPP*Y*, bReg*Y*, and bR*Y*", where Y ranges from 1 to 3.

Table 3 gives the measurement of our benchmark using the standard collection library of Pharo. This table is used as the baseline for our improvements of the library.

Minimizing measurement bias. Carefully considering measurement bias is important since an incorrect setup can easily lead to a performance analysis that yields incorrect conclusions. Despite numerous available methodologies, it is known that avoiding measurement bias is difficult [8, 15]. An effective approach to minimize measurement bias is called *experimental setup randomization* [15], which consists in generating a large number of experimental settings by varying some parameters, each considered parameters being a potential source of measurement variation. Our measurements are programmatically triggered, meaning that multiple runs of our benchmark is easily automatized. We have considered the following parameters:

- *Hardware and OS* – We have used two different hardwares and operating systems ((a) a MacBook Air, 1.3Ghz Intel Core I5, 4Gb 1333 MHz DDR3, with a solid hard disk and (b) iMac, Quad-core Intel Core i5, 8 Gb).
- *Heap size* – We run our experiment using different initial size of the heap (20Mb, 150Mb, 300Mb).
- *Repeated run* – For each execution of the complete benchmark, we have averaged 5 runs, with a random pause between each run.
- *Randomized order* – The individual benchmarks (*i.e.,* a unit-test benchmark or a performance benchmark) are randomized at each complete benchmark run.
- *Reset caches* – Method cache located in the VM are emptied before each run.
- *GC* – Garbage collector has been activated several times before running each benchmark.

In total, we have considered 9 different experimental setups. We did not notice any significant variation between these experimental setups.

The measurements given in appendix are the result of an average of 9 different executions, each considering a different combination of the parameters given above.

4. Use of Expandable Collections in Pharo applications

This section analyzes the use of expandable collections in Pharo applications. The results given in this section answer the research question A.

4.1 Dynamic analysis

We have run our two sets of our benchmark and profiled their executions. The metrics given in Section 3.2 have been

computed and reported in Table 3 for each of the applications execution.

The execution of the 17 unit test benchmarks create a total of 2,474,499 expandable collections and the 15 performance benchmarks create 6,342,087 expandable collections. The analyses this paper describes focus on the profiling of nearly 9M expandable collections produced by 32 different program executions (17 unit test benchmarks + 15 performance benchmarks).

Naturally, very few of these expandable collections live through the whole execution since the garbage collector regularly cleans the memory by removing unreferenced collections. In our measurements, we do not consider the action of the garbage collector on the collection themselves since garbage collection is orthogonal to the research questions we are focusing on.

The number of created collections indicates large disparities between the analyzed applications. Benchmarks 1, 10, bReg1 and bReg2 involve a long and complex execution over a significant amount of data, indicated by the large number of created expandable collections. Benchmarks 2, 6, 8, bN1, bN2, bN3 create a small number of collections, indicating short executions.

Variation in the measurements. Two executions of the same code may not necessarily create the same number of collections, even if no input/output or random number generation is involved. Measurements vary little over multiple runs of the benchmarks. Values reported in the tables in the appendix have been obtained after 10 runs and have an average variation of 0.0095%. Although the applications we have selected for our case study do not make use of random number generation, the use of hash values can make non deterministic behavior. A hash value is given by the virtual machine when the object is created. In the case of Pharo, such a hash value depends on an internal counter of the virtual machine. Consider the following code:

```
d := Dictionary new.
d at: key1 put: OrderedCollection new.
d at: key2 ifAbsentPut: [ OrderedCollection new ]
```

The class `Dictionary` uses the equality relation and hash values between keys to insert pairs. If we have the relation `key1 = key2` and `key1 hash = key2 hash`, then the dictionary considers that the two keys are actually the same and we have only one instance of `OrderedCollection`. However, in case that the `hash` is not overridden but `=` is overridden, the relation `key1 hash = key2 hash` may be true only sporadically, thus triggering a non deterministic behavior over multiple executions[3].

Empty collections. Table 3 indicates a surprisingly high proportion of empty collections in our benchmarks. From over 8.6 million expandable collections created by our benchmarks, 6.6 million (76%) have been created without having any element added to them. Only 23% of collections have at least one element added to them during their lifetime.

To understand this phenomena better, we will take a closer look at the data we obtained. The number of empty collections created by our benchmark varies significantly across applications. Consider the application 10 and its corresponding benchmark. Benchmark 10 creates a total of 1.4M of expandable collections, for which only 14,891 are non-empty. This application is a refactoring engine that applies pattern matching and rewriting rules on source code. The engine is complex due to the underlying optimized logic engine[4]. By excluding this application, the ratio of the number of not empty collections for the unit test benchmarks rises to 31.3% (*NC*= 1,043,634 and *NNEC*= 327,445): about one-third of collections created by the unit tests are left empty in the average.

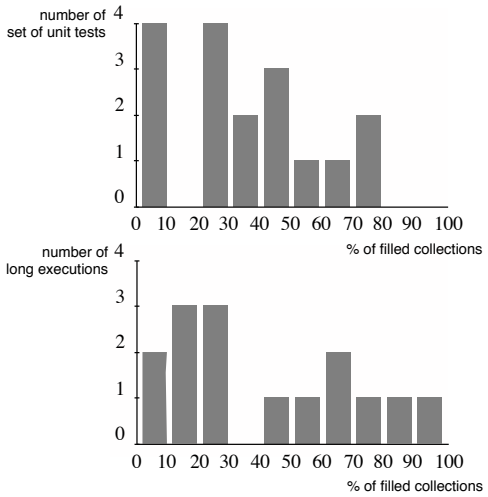


Figure 1: Frequency distribution of filled collections (*NNEC*)

Figure 1 shows the frequency distribution of the benchmark for both unit tests and performance benchmarks. There are four applications that have less than 10% of collections empty, and four applications that have between 20% and 30% of collections that are not empty. Performance benchmarks have a tendency to fill more expandable collections compared with unit test benchmarks. This highlight an important difference between two unit-test benchmarks and performance benchmarks. The first has tendency to create many empty collections over the latter. Benchmark bN3 generates no empty collection.

Cause of empty collections. We manually have inspected the applications and benchmarks that generate a high proportion

[3] Redefining `=` without redefining `hash` is a classic defect in software programs and it is widely recognized as such. Unfortunately, this defect is frequent.

[4] Interestingly, PMD, a Java application similar to Refactoring, exhibits the very same problem [6].

of empty collections. A large proportion of the created empty collections is caused by the object initialization specified in the constructors. Consider the constructor of the class `RBVariableEnvironment`:

```
RBVariableEnvironment >> initialize
  super initialize.
  instanceVariables := Dictionary new.
  classVariables := Dictionary new.
  instanceVariableReaders := Dictionary new.
  instanceVariableWriters := Dictionary new
```

This constructor implies that each instance of `RBVariable Environment` comes with at least four instances of dictionaries. The class `RBVariableEnvironment` is part of a code meta-model that belongs to the application `Refactoring`. Most instances of `RBVariableEnvironment` actually have their dictionaries empty, which contributes to the 98% of the collections created by Benchmark 10 being left empty. This is not an isolated case. The 17 applications under study are composed of 1,713 classes. We have 375 of these 1,713 classes that explicitly define at least one constructor. We have also found that 144 of these 375 classes explicitly instantiate at least one expandable collection when being instantiated.

Expandable collections created in the constructor and lazy evaluation of variable are a prominent cause of unused collections.

Number of array creations. The standard collection library creates a new array at each collection expansion. Since instantiating a collection results in creating a new array, the number of created arrays (*NAC*) subtracted to the number of expansions (*NCE*) is equal to the number of collections (*NC*). We have roughly the following relation $NAC - NCE = NC$ in Table 3. Some differences may be noticed due to rehashing operations on hash-based collections (*e.g.,* HashSet, Dictionary) that may be triggered by an application. Such effects are marginal and have a little impact on the overall measurements, which is why we do not investigate such minor variations further.

Collection expansions. From the 8.6M of collections (*NC* column), only 0.56% of the collections are expanded 1,002,212 times during the execution of the benchmark (*NCE* column). These expansions result in over 62Mb of copies between these arrays (*NCB* column).

Unused memory. Summing up the memory consumed by all the internal arrays yields over 342Mb. More than 296Mb of these 342Mb are actually unused as a result of having expandable collections filled only a little on average (*i.e.,* the size of the collection being much below its capacity).

4.2 Reducing the overhead incurred by collection expansions

The measurements given in the previous section reveal that the use of expandable collections may result in wasted CPU and memory consumption. We use the observations made above to reduce the overhead caused by expansions. We propose three heuristics to reduce the overhead incurred by expandable collections:

Creating the internal array storage on demand. Creating an internal array only when necessary, *i.e.,* at the first element added. Since 76% of arrays are empty, lazily instantiating the internal array will be beneficial.

Reusing arrays when expanding. Expanding a collection involves creating an array larger than the previous one (usually twice the initial size). After copying, the original array is discarded by removing all references to it. The task to free the memory is then left to the garbage collector.

Instead of letting the garbage collector discard old arrays, arrays can be recycled: a collection expansion frees an array, which itself may be used when another collection expands.

Setting an initial capacity. About 10% of expandable collections are created and filled in the same method. These 10% of the collections have been created by 276 methods across our benchmark. There are 105 of these 276 methods that use the default construction with the default initial capacity.

Some of these methods may be refactored to create expandable collections with an adequate initial capacity.

We have conceived the *OptimizedCollection* library, a collection library for Pharo that exhibits better resource management than the standard set of collection classes. OptimizedCollection implements the design points made above. Section 5, Section 6 and Section 7 elaborate on each of these points.

5. Lazy Internal Array Creation

In Pharo, expandable collections have been implemented under the assumption that a collection will be filled with elements. This assumption unfortunately does not hold for the usage scenarios we are facing in our benchmark. Less than a third of the expandable collections are filled in practice. This suggests that creating the internal array only when elements are added is likely to be beneficial. We call this mechanism *lazy internal array creation*.

This section describes the first design point, which is to support lazy internal array creation.

5.1 Creating the array only when necessary

Introducing a lazy creation of the internal array is relatively easy to implement. Instead of creating the internal storage in the constructor, we defer its creation when adding an element to the collection. For this, we need to remember the capacity for the future creation of the array. Methods that add elements to the collection have to be updated accordingly.

This simple-to-implement improvement leads to a significant reduction in memory consumption. Using the default capacity, an empty ordered collection now occupies 20 bytes only (in comparison with the 64 bytes without supporting lazy internal array creation). After adding an element to the

collection, the internal array is created, thus increasing the size of the collection to 64 bytes.

We have implemented the lazy internal array creation as described above in all the expandable collection classes. The following section describes the impact on our case studies.

5.2 Lazy creation on the benchmark

Table 4 gives the metric values of our benchmark when using the lazy internal array creation. Contrasting Table 3 (using the standard collection library, *i.e.,* without lazy internal array creation) with Table 4 (lazy creation) shows a significant reduction of unused memory and number of created internal arrays. More specifically, we have:

- The number of array creation (*NAC*) has been significantly reduced as one would expect. It went from 8,701,783 down to 2,437,083, representing a reduction of (8,605,147 − 2,437,083) / 8,605,147 = 71.67% of array creation.
- The number of unused bytes (*NUB*) has also been significantly reduced. It went from 296Mb down to 82Mb, representing a reduction of (296 − 82) / 296 = 72.29%.

Application 10 is producing a high number of expandable collections that remains empty during the overall execution. Using the original collection library, Application 10 created 1,438,380 internal arrays (*NAC* column in Table 3). Making the collection library support the lazy internal array creation makes this value goes to 68,098. A reduction of (1,438,380 − 68,132) / 1,438,380 = 95% of created arrays.

The lazy internal array creation has a slight positive impact on the execution time of the benchmark. By lazily creating the internal arrays, the execution time has been reduced by 2.38%.

6. Recycling Internal Arrays

A collection expansion is carried out with three sequential steps (Section 2.1): (i) creation of a larger array; (ii) copying the old array into the new one; (iii) replacing the collection's storage with the new array. The third step releases the unique reference of the array storage, entitling the array to be disposed by the garbage collector. This section is about recycling unused internal arrays and measures the benefits of recycling.

The general mechanism of recycling arrays along a program execution is not new. It has already been shown that for functional programming avoiding unnecessary array creation by recycling those arrays is beneficial [17]. Recycling arrays in a context of expandable collections is new and, as far as we are aware of, it has not been investigated.

6.1 Recycling arrays on the benchmark

Principle. Instead of releasing the unique reference of an array, the array is recycled by keeping it within a globally accessible pool. The array disposed after a collection expansion is inserted in the pool. The first step of expansion has now to check for a suitable array from the pool. If a suitable array is found, the array is removed from the pool and used as internal array storage in the expanded collection. If no array from the pool can be used as internal array storage for a particular collection expansion, a new array is created following the standard behavior.

When an array is inserted into the pool, the array has to be emptied so as to not keep unwanted references. Emptying an array is done by filling it with the `nil` value.

Need for different strategies. Consider the following example:

```
c1 := OrderedCollection new.
50 timesRepeat: [ c1 add: 42 ].
c2 := OrderedCollection new.
c3 := OrderedCollection new.
```

Filling `c1` with 50 elements triggers three expansions, which increases the capacity from 10 to 20, from 20 to 40 and from 40 to 80. Having `c1` of a capacity of 80 is sufficient to contain the 50 elements. The creation of the collection and these expansions has created and released three arrays sized 10, 20, 40, respectively. These arrays are inserted in a pool of arrays.

When `c2` is created, an array of size 10 is needed for its internal array storage. The pool of arrays contains an array of size 10 (obtained from the expansion of `c1`). This array is therefore removed from the pool and used for the creation of `c2`.

Similarly, `c3` requires an array of size 10. The pool contains two arrays, of size 20 and size 40. The creation of the ordered collection faces the following choice: either we instantiate a new array of size 10, or we use one of the two available arrays.

This simple example illustrates the possibility of having different strategies for picking an array from the pool. We propose three strategies and evaluate their impact over the benchmark:

S1: *requiredSize* = *size* – Pick an array from the pool of exactly the same size that is requested

S2: *requiredSize* <= *size* – Pick the first array with a size equal to or greater than what is requested

S3: *size / 0.9* < *requiredSize* < *size * 1.1* – Pick an array which has a size within a range of 20% of what is requested.

The effects of the different strategies on the unit-test benchmark is summarized in Table 1. We consider 8 metrics: *NC* (number of created expandable collections), *NCE* (number of collection expansions), *NCB* (number of copied bytes), *NAC* (number of internal array creations), *NAB* (number of allocated bytes), *NUB* (number of unused bytes), the number of full garbage collections and the number of incremental garbage collections.

S1 generates less unused array portions (*NUB*) than *S2* and *S3*. *S2* incurs less collection expansions than *S1* and *S3*,

metrics	*S1*	*S2*	*S3*
NC	2,475,658	2,475,670	2,475,708
NCE	21,134	**19,118**	21,139
NCB	15,519,656	**14,761,492**	15,544,560
NAC	**542,622**	542,757	542,863
NAB	31,467,464	36,983,740	**31,445,140**
NUB	**21,332,420**	26,808,248	21,334,960
#full GC	40	44	40
#incr GC	14,442	**14,423**	20,314

Table 1: Effect of the different strategies for the unit test benchmarks (best performance is indicated in **bold**)

which also result in fewer copied bytes (*NCB*). Oddly, the number of incremental garbage collections is higher with *S3*.

Effect on the benchmark. Table 5 details the use of strategy *S1* on the benchmark. When supporting the lazy internal array creation without recycling arrays (Table 4), the number of unused bytes has been reduced by $(82,927,120 - 82,752,904)/82,927,120 = 0.2\%$. The reduction of the number of created arrays is $(2,437,083 - 2,341,191)/2,437,083 = 4\%$. In all, 35,063 collections have been recycled. More interestingly, the technique of reusing arrays has reduced the number of allocated bytes by 14.6% (column *NAB* : $(128,678,564 - 109,840,556)/128,678,564 = 14.6\%$).

After profiling the benchmark, the number of collections left over in the pool is rather marginal. Only 216 collections are in the pool, totaling less than 89kB.

Using the pool of arrays incurs a relatively small execution time penalty. This represents an increase of 5.8% of execution time when compared with the lazy array creation and an increase of 2.8% with the original library.

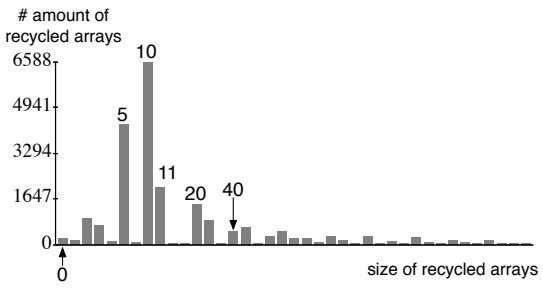


Figure 2: Distribution of recycled arrays

Recycled arrays. The techniques described in this section recycle arrays of different sizes. Figure 2 shows the distribution of size of recycled arrays for Strategy $S1$. The vertical axis indicates the number of recycled arrays. The horizontal axis lists the size of arrays that are effectively recycled.

Arrays that are the most recycled have a size of 5 and 10. The standard Pharo library is designed as follows: 5 corresponds to the minimum capacity of hash-based collections, and 10 is the default size of non-hashed collections[5]. The value 20 corresponds to the size of the internal array of a default collection after expansion. An array of size 40 is obtained after a second expansion.

Multi-threading. The pool of recycled internal array is globally accessible. Accesses to the pool need to be adequately guarded by monitors to avoid concurrent addition or removal from the pool. Several of the applications included in our benchmark are multi-threaded. However, the execution scenarios we consider are not thread-intensive. Previous work on pooling reusable collections [18] shows satisfactory performance in a multi-threaded setting.

6.2 Variation in time execution

If we consider the global figures, recycling arrays has a penalty of 3% of execution time in the average. However, if we have a close look at each individual benchmark, we see that most of the performance variation indicates that our optimized collection library performs slightly faster than the standard collection library (in addition to significantly reduce the memory consumption, as detailed in the previous sections).

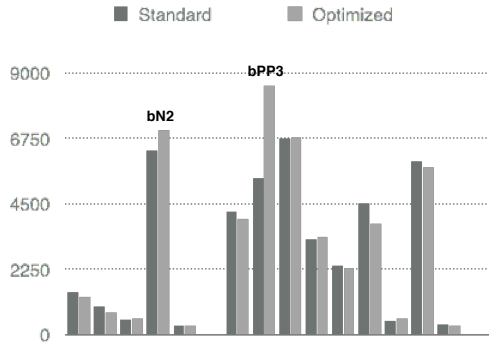


Figure 3: Impact of execution time of the optimized collection library

Figure 3 shows the variation of execution time of the performance benchmarks between the standard collection library and our optimized library. All but two benchmarks are slightly faster with our library. The execution of benchmarks *bN2* takes 6,738 seconds with the standard collection library and takes 6,789 with our library. Since this represents a variation of $(6,789 - 6,738)/6,738 = 0.7\%$, we consider this variation as insignificant.

[5] Note that we are not arguing whether 5 and 10 are the right default size. Other languages including Scala and Ruby use a different default capacity size. We are simply considering what the Pharo collection library offers to us.

Benchmark *bPP3* goes from 6,330 seconds with the standard library to 7,010 with our optimized library, which represents an increase of 9.7%. The reason for this drop in performance is not completely clear to us. This benchmark parses a massing amount of textual data. Private discussion with the authors of the considered application revealed the cause of this variation may be due to the heavy use of short methods on streams. Traditional sampling profiler does not identify the cause of the performance drop, which indicates us that its stems from particularities of the virtual machine (for which its execution is not captured by the standard Pharo profiler). These short methods have an execution time close to the elementary operations performed by the virtual machine to lookup the message in method cache. Although we carefully designed our execution by emptying different caches and multiply activating the garbage collection between each execution, the reason of the performance drop may be related to some particularities of the cache in the virtual machine.

By excluding the benchmark *bPP3*, our library performs 3.01% faster than with the standard collection library. When considering this outlier, our performance benchmark runs 7.9% slower.

7. Setting Initial Capacities

A complementary approach to improving the collection library is to find optimization opportunities in the base application (which makes use of the collection library).

Example. We have noticed recurrent situations for which an expandable collection is filled in the same method that creates the collection. The following method, extracted from a case study, illustrates this:

```
ROView>>elementsToRender
  | answer |
  answer := OrderedCollection new.
  self elementsToRenderDo: [ :el | answer add: el ].
  ^ answer
```

The method `elementsToRender` creates an instance of the class `OrderedCollection` and stores it in a temporary variable called `answer`. This collection is then filled by iterating over a set of elements.

The method `elementsToRender` uses the default constructor of the class `OrderedCollection`, which means a default capacity to the collection is given. As described in the previous sections, such a method is a possible source of wasted memory since a view may contain a high number of elements, thus recreating the situation we have seen with the micro-benchmark in Section 2.1.

By inspecting the definition of the method `elementsToRenderDo:`, we have noticed that the number of elements to render is known at that stage of the execution. The method may be rewritten as:

```
ROView>>elementsToRender
  "Return the number of elements that will be rendered"
  | answer |
  answer := OrderedCollection new: (self elements size).
  self elementsToRenderDo: [ :el | answer add: el ].
  ^ answer
```

This new version of `elementsToRender` initializes the ordered collection with an adequate capacity, meaning that no resource will be wasted due to the addition of elements in the collection referenced by `answer`.

Profiling. The metrics *NOSM* and *NSM* identify methods that create a collection and fill it. The instance of `OrderedCollection` created by the method `elementsToRender` is counted by *NSM* since the collection is created and filled in this method. The collection is also counted by *NOSM* in the case that no other methods add or remove elements from the result of `elementsToRender`.

We see that about 8% of the expandable collections are immediately filled after their creation. We also notice that slightly fewer collections are only filled in the same method in which they were created. We are focusing on these collections since they are likely easy to refactor without requiring a deep knowledge about the application internals.

The *NOSM* and *NSM* metrics are computed by instrumenting all the constructors of expandable collection classes and all the methods that add and remove elements.

Refactoring methods. The 204,680 collections that are filled solely in the methods that have created them have been produced by exactly 276 methods. We have manually reviewed each of these methods. We have refactored 105 of the 276 methods to insert a proper initialization of the expandable collection. The remaining 171 methods were not obvious to refactor. Since we did not author these applications and had a relatively low knowledge about the internals of the analyzed applications, we took a conservative approach: we have refactored only simple and trivial cases for which we had no doubt about the initial capacity, as in the example of `elementsToRender` given above. We use unit-test to make sure we did not break any invariant captured by the tests.

Impact on the benchmark. Table 6 details the profiling for the benchmark by lazily creating internal arrays, reusing these arrays and refactoring the applications. The reduction gain for the number of allocated bytes is 0.11% (column *NAB*, which goes from 109.840Mb to 109.713Mb). The amount of unused space has been reduced by 0.12% (column *NUB*, which goes from 82.752Mb down to 82.651Mb). No variation in terms of execution time has been found.

Setting the capacity. We have run the *modified version of our benchmark* with the *original collection library*, without the recycling and the lazy array creation. Again, gains are marginal. Only a reduction of 0.13% of the number of allocated bytes has been measured.

We conclude that the obtained gain by allocating a proper initial capacity is marginal.

8. Other programming languages

This section reviews four programming languages (Java, C#, Scala, and Ruby) by briefly describing how collections are handled in these and how our result may be applied to them.

Java. The Java Collection Framework is composed of 10 generic interfaces implemented by 10 classes. In addition, the framework offers 5 interfaces for concurrent collections. We restrict our analysis to general purpose collections, however.

Classes describing collections are very similar to Pharo's. For example[6], the class `ArrayList` uses an internal array to store elements, as `OrderedCollection` does. The class `HashSet` wraps an instance of `HashMap`. `HashMap` uses an array of `Entry` elements, each entry being an association *(key, value)*. The implementation of `HashSet` is again very similar to Pharo's `Dictionary`, with a 0.75 threshold to trigger an expansion. The class `TreeMap` does not have an equivalent in standard Pharo and uses an array to store a collection's elements.

In Pharo we did not consider the class `LinkedList` since this class is only used by the runtime and not by user-defined applications. However, in Java, `LinkedList` is used more and other collection classes are built on it, *e.g.,* `LinkedHashSet` and `LinkedHashMap`.

C#. `ArrayList` is similar to its Java sibling and Pharo's `OrderedCollection`. The C# version of `ArrayList` initializes its internal array with an empty array, resulting in an implementation equivalent to the lazy internal array creation (Section 5).

`Hashtable` uses an internal array which is created with the proper capacity when the class is instantiated. `Hashtable` does not use an empty array as `ArrayList` does. The class `Dictionary` and `Queue` do not lazy initialize its internal array storage. Similarly to `ArrayList`, `Stack` initializes its internal array storage with an empty list, thus triggering an expansion at the first element addition.

Scala. Instead of simply wrapping Java collections as many languages do when running on top of the Java Virtual Machine, Scala offers a rich trait-based collection library that supports statically checked immutability [19] (which Java does not support). The design of expandable collections in Scala is similar to Java. `ArrayBuffer` which is the equivalent of Java's `ArrayList` creates an empty array of a default size 16.

`ArrayBuffer` extends `ResizeableArray`[7] utility class used by several other collections. The private array field in that class is called `array` and the logic of manipulating it is the same as with Java's `ArrayList`.

Ruby. Oddly, Ruby provides the complete implementation of array, the most used expandable collection in Ruby, in the virtual machine. All the arithmetic operations, copy, element addition and removing are carried out by the virtual machine. Ruby associates to each empty collection an array of size 16.

Applicability of our results. In our experiment we have identified a significant amount of empty collections. Similar behavior has been found in other situations. For example, when conducting the case studies in Java with Chameleon [6], a high proportion of empty collections have also been identified.

The collection framework of Java[8], C#, Scala, and Ruby behave similarly to Pharo, except for the C# version of `ArrayList` and `Stack`. We therefore expect our improvement on the Pharo library to have a positive and significant impact on these collection libraries. As future work, we plan to verify assumption by modifying the standard library and running established benchmarks (e.g., DaCapo [13]).

9. Related Work

Patterns of memory inefficiency. A set of recurrent memory patterns have been identified by Chis *et al.* [12]. Overheads in Java come from object headers, null pointers, and collections. Three of their 11 patterns (P1, P3, P4) are about unused portions internal arrays of collections. The model *ContainerOrContained* has been proposed to detect occurrences of these patterns.

We have proposed the lazy internal array creation technique to efficiently address pattern *P1 - empty collections*. Addressing pattern *P3 - small collections* is unfortunately not easy. Our collection profiler identifies the provenance of collections having an unnecessary large capacity. However refactoring the base application to properly set the capacity does not result in a significant impact (only a reduction of 0.13% of allocated bytes has been measured). As future work, we plan to verify whether some patterns, depending on the behavior of the application, may be identified (*e.g.,* a method that always produce collections of a same size).

Storage strategies. Use of primitive types in Python may trigger a large number of boxing and unboxing operations. Storage strategies [20] significantly reduce the memory footprint of homogeneous collections. Each collection has a storage strategy that is dynamically chosen upon element additions. Homogeneous collections use a dedicated storage to optimize the resources consumed by the storage.

Storage strategies may be considered as a generalization of the lazy internal array creation described above. Our approach focuses on reducing the memory footprint of expandable collections, which is different, but complementary to the approach of Bolz, Diekmann and Tratt which focuses on the representation in memory of homogenous collections.

Discontiguous arrays. Traditional implementation of memory model uses continuous storage. Associating a continuous

[6] The source code of `ArrayList` is visible online on http://bit.ly/ArrayListOpenJDK6

[7] https://github.com/scala/scala/blob/ master/src/library/scala/collection/-mutable/ResizableArray.scala

[8] A private discussion with some developers at Oracle indicates that an updated version of the Collection library in JDK 7 will soon support lazy array creation.

memory portion to a collection is known to be a source of wasted space which leads to unpredictable performance due to garbage collection pauses [21]. *Discontiguous arrays* is a technique that consists in dividing arrays into indexed memory chunks [10, 11, 22, 23]. Such techniques are particularly adequate for real-time and embedded systems.

Implementing these techniques in an existing virtual machine usually comes at a heavy cost. In particular, the garbage collector has to be aware of discontiguous arrays. A garbage collector is usually a complex and highly optimized piece of code, which makes it it very delicate to modify. Bugs that may be inadvertently introduced when modifying it may result in severe and hard-to-trace crashes.

Our results show that a significant improvement may be carried out without any low-level modification in the virtual machine or in the executing platform. Many of our experiments about memory profiling in Pharo have been carried out having simultaneously multiple different versions of the collection library. Nevertheless, research results about discontinuous arrays, in particular Z-rays [11], may be beneficial to expandable collections. In the future, we plan to work on this.

Dynamic adaptation. Choosing the most appropriate collection implementation is not simple. The two collections `ArrayList` and `HashSet` are often chosen because their behavior is well known, which makes them popular. Improperly chosen collection implementation may lead to unnecessary resource consumption. Xu [5] proposes an optimization technique to dynamically adapt a collection into the one that fits best according to its usage (*e.g.,* replacing a `LinkedList` with an `ArrayList`).

Xu's approach is similar to the storage strategies mentioned above, which makes it complementary to our approach.

Adaptive selection of collections. In the same line as dynamic adaption, Shacham *et al.* [6] describe a profiler specific to collections which outputs a list of appropriate collection implementation. The correction can be either made automatically, or presented to the programmer for correction. A small domain-specific language is described to define rules to characterize use of collections.

Recycling collections. The idea of recycling some collections classes has been investigated in the past. For example, functional languages create a new copy, at least in principle, at each element addition or removal. Avoiding such copies has been the topic of numerous research work [17, 24].

Recycling collections when possible is known to be effective [25]. For example, *Java Performance Tuning* [18], Chapter 4, Page 79, mentions "Most container objects (e.g., `Vector`s, `Hashtable`s) can be reused rather than created and thrown away." However, no evidence about the gain is given. In the case of Pharo, recycling internal arrays of expandable collections reduces the number of allocated bytes by 14.6%. This chapter also argues that recycling collections is effective in a multi-threaded setting. Although our benchmarks includes multi-threaded applications, our execution did not make an heavy use of threads. This book chapter supports the idea that programmers should make their collection reusable, whenever is possible. Our work embeds this notion of recycling arrays within the collection library itself.

The notion of redundant computation within loops has been the topic of some recent work [26–28]. Efficient model for reusing objects at loop iteration are provided. For example, reusing collections within loop leads to a "20-40% reduction in object churn" and "the execution time improvements range between 6-20%." Object churn refers to the excessive generation of temporary objects. Our approach essentially embeds the improvement within the collection library, which has the advantage to not impact the programmer's habits. However, our performance improvement are lesser.

Adaptive collection. The Clojure programming language[9] offers persistent data structures. Such data structures have their implementation based on the usage of the internal array storage. For example, a `PersistentArrayMap` is promoted to a `PersistentHashMap` once the collection exceeds 16 entries.

10. Conclusion and Future Work

Expandable collections are an important piece of the runtime. Although intensively used, expandable collections are a potential source of wasted memory space and CPU consumption.

Improving the performance of expandable collections went through three different steps, as described in Section 5, Section 6 and Section 7. We have defined a total of 32 executions of 17 different applications, which generate nearly 9M of expandable collections. The execution blueprint of these collections obtained with the standard collection library is given in Table 3. We have developed OptimizedCollection, a collection library that supports lazy array creation and array recycling. The execution profile of the benchmark is given in Table 5. The positive effect of our collection is given by contrasting Table 5 against Table 3. OptimizedCollection has:

- reduced the number of created intermediary internal array storage by (8,701,783 − 2,341,191) / 8,701,783 = 73.09% (column *NAC*)
- reduced the number of allocated bytes by (342,818,892 − 109,840,556) / 342,818,892 = 67.95% (column *NAB*)
- reduced the number of unused bytes by (296,863,696 − 82,752,904) / 296,863,696 = 72.12% (column *NUB*)

Recycling arrays incur a time penalty on the execution. Our benchmark runs 3% faster for all but one performance benchmark.

Acknowledgments. We thank Oscar Nierstrasz, Lukas Renggli, Eric Tanter, and Renato Cerro for their comments on an early draft of this paper. We also thank Aleksandar Prokopec for his help with Scala collections.

[9] `http://clojure.org`

References

[1] W. R. Cook, On understanding data abstraction, revisited, SIGPLAN Not. 44 (10) (2009) 557–572.

[2] K. Wolfmaier, R. Ramler, H. Dobler, Issues in testing collection class libraries, in: Proceedings of the 1st Workshop on Testing Object-Oriented Systems, ETOOS '10, ACM, New York, NY, USA, 2010, pp. 4:1–4:8.

[3] S. Ducasse, D. Pollet, A. Bergel, D. Cassou, Reusing and composing tests with traits, in: Tools'09: Proceedings of the 47th International Conference on Objects, Models, Components, Patterns, Zurich, Switzerland, 2009, pp. 252–271.

[4] J. Y. Gil, Y. Shimron, Smaller footprint for java collections, in: Proceedings of OOPSLA'11, pp. 191–192.

[5] G. Xu, Coco: Sound and adaptive replacement of java collections, in: Proceedings of ECOOP'13, pp. 1–26.

[6] O. Shacham, M. Vechev, E. Yahav, Chameleon: Adaptive selection of collections, in: Proceedings of PLDI '09, pp. 408–418.

[7] D. Cassou, S. Ducasse, R. Wuyts, Traits at work: the design of a new trait-based stream library, Journal of Computer Languages, Systems and Structures 35 (1) (2009) 2–20.

[8] T. Kalibera, R. Jones, Rigorous benchmarking in reasonable time, in: Proceedings of the 2013 International Symposium on Memory Management, ISMM '13, ACM, New York, NY, USA, 2013, pp. 63–74.

[9] S. Wilson, J. Kesselman, Java Platform Performance, Prentice Hall PTR, 2000.

[10] S. Joannou, R. Raman, An empirical evaluation of extendible arrays, in: Proceedings of the 10th International Conference on Experimental Algorithms, SEA'11, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 447–458.

[11] J. B. Sartor, S. M. Blackburn, D. Frampton, M. Hirzel, K. S. McKinley, Z-rays: Divide arrays and conquer speed and flexibility, in: Proceedings of PLDI '10, pp. 471–482.

[12] A. E. Chis, N. Mitchell, E. Schonberg, G. Sevitsky, P. O'Sullivan, T. Parsons, J. Murphy, Patterns of memory inefficiency, in: Proceedings of ECOOP'11, pp. 383–407.

[13] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, B. Wiedermann, The dacapo benchmarks: java benchmarking development and analysis, in: Proceedings of OOPSLA '06, pp. 169–190.

[14] R. C. Martin, Agile Software Development. Principles, Patterns, and Practices, Prentice-Hall, 2002.

[15] T. Mytkowicz, A. Diwan, M. Hauswirth, P. F. Sweeney, Producing wrong data without doing anything obviously wrong!, in: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems, ASPLOS '09, ACM, New York, NY, USA, 2009, pp. 265–276.

[16] M. Dias, M. M. Peck, S. Ducasse, G. Arévalo, Fuel: a fast general purpose object graph serializer, Software: Practice and Experience 44 (4) (2014) 433–453.

index	Application	LOC	#Ref
*1	AST	8,091	57
2	Arki	627	6
3	Glamour	17,525	105
4	GraphET	2,757	10
5	Magritte	5,884	29
6	Manifest	2,864	11
7	NECompletion	3,446	33
*8	Nautilus	1,566	9
*9	Petit	14,919	95
10	Refactoring	21,328	125
*11	Regex	5,055	16
12	Ring	3,378	50
*13	Roassal	19,844	133
14	RoelTyper	2,003	85
15	Shout	3,290	10
16	SmallDude	3,805	66
17	Spec	10,212	37

Table 2: Description of the benchmark (the #Ref column indicates the number of references to expandable collection in source code)

[17] A. Kagedal, S. Debray, A practical approach to structure reuse of arrays in single assignmentlanguages, Tech. rep., Tucson, AZ, USA (1996).

[18] J. Shirazi, Java Performance Tuning, 2nd Edition, O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[19] M. Odersky, A. Moors, Fighting bit rot with types (experience report: Scala collections), in: R. Kannan, K. N. Kumar (Eds.), IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2009)

[20] C. F. Bolz, L. Diekmann, L. Tratt, Storage strategies for collections in dynamically typed languages, in: Proceedings of OOPSLA '13, pp. 167–182.

[21] P. Wilson, M. Johnstone, M. Neely, D. Boles, Dynamic storage allocation: A survey and critical review, in: H. Baler (Ed.), Memory Management, Vol. 986 of LNCS, 1995, pp. 1–116.

[22] D. F. Bacon, P. Cheng, V. T. Rajan, A real-time garbage collector with low overhead and consistent utilization, in: Proceedings of POPL '03, pp. 285–298.

[23] G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, B. Mathiske, M. Wolczko, Heap compression for memory-constrained java environments, in: Proceedings of OOPSLA '03, pp. 282–301.

[24] N. Mazur, P. Ross, G. Janssens, M. Bruynooghe, Practical aspects for a working compile time garbage collection system for mercury, in: P. Codognet (Ed.), Logic Programming, Vol. 2237 of LNCS, 2001, pp. 105–119.

[25] G. Xu, Finding reusable data structures, in: Proceedings of OOPSLA '12, pp. 1017–1034.

[26] S. Bhattacharya, M. G. Nanda, K. Gopinath, M. Gupta, Reuse, recycle to de-bloat software, in: Proceedings of ECOOP'11, pp. 408–432.

[27] G. Xu, D. Yan, A. Rountev, Static detection of loop-invariant data structures, in: Proceedings of ECOOP'12, pp. 738–763.

[28] A. Nistor, L. Song, D. Marinov, S. Lu, Toddler: Detecting performance problems via similar memory-access patterns, in: Proceedings ICSE '13, pp. 562–571.

A. Application Benchmark Detail

B. Measurement

bench.	*NC*	*NNEC*	*NEC*	*NCE*	*NCB*	*NAC*	*NOSM*	*NSM*	*NAB*	*NUB*
1	643,603	151,665(23%)	491,938(76%)	2,636	35,940	646,239	112,308(17%)	112,439(17%)	18,819,920	17,907,204
2	13	12(92%)	1(7%)	0	0	13	1(7%)	1(7%)	468	392
3	78,976	34,912(44%)	44,064(55%)	322	2,272	79,298	10,604(13%)	10,681(13%)	2,374,464	2,213,384
4	1,690	512(30%)	1,178(69%)	30	1,720	1,720	245(14%)	245(14%)	65,760	55,420
5	3,191	2,009(62%)	1,182(37%)	30	1,208	3,193	200(6%)	251(7%)	77,808	67,724
6	96	44(45%)	52(54%)	0	0	96	40(41%)	40(41%)	3,740	3,464
7	612	218(35%)	394(64%)	614	2,257,960	1,231	48(7%)	48(7%)	4,657,696	837,948
8	2	0(0%)	2(100%)	0	0	2	0(0%)	0(0%)	40	40
9	158,589	58,371(36%)	100,218(63%)	5,663	280,876	164,252	57,644(36%)	57,728(36%)	6,833,532	5,534,280
10	1,432,306	14,891(1%)	1,417,415(98%)	6,074	12,258,424	1,438,380	8,248(0%)	8,344(0%)	46,958,632	34,241,204
11	6,839	2,058(30%)	4,781(69%)	1,280	78,712	6,967	471(6%)	480(7%)	291,052	256,852
12	8,363	3,530(42%)	4,833(57%)	1,103	47,852	9,466	73(0%)	73(0%)	279,236	165,460
13	108,571	57,590(53%)	50,981(46%)	1,739	369,336	109,990	8,151(7%)	8,810(8%)	5,778,016	4,845,764
14	10,305	586(5%)	9,719(94%)	145	14,044	10,448	82(0%)	110(1%)	443,828	420,352
15	20,815	14,886(71%)	5,929(28%)	255	125,900	21,070	5,736(27%)	5,740(27%)	2,692,404	1,984,240
16	766	172(22%)	594(77%)	17	496	783	123(16%)	123(16%)	126,368	122,532
17	1,203	880(73%)	323(26%)	1,512	48,384	2,715	764(63%)	764(63%)	127,356	35,988
total	**2,475,940**	**342,336(13%)**	**2,133,604(86%)**	**21,420**	**15,523,124**	**2,495,863**	**204,738(8%)**	**205,877(8%)**	**89,530,320**	**68,692,248**
bAST1	210,000	38,000(18%)	172,000(81%)	0	0	210,000	38,000(18%)	38,000(18%)	6,752,000	6,468,000
bAST2	179,000	47,000(26%)	132,000(73%)	4,000	24,000	183,000	41,000(22%)	41,000(22%)	5,928,000	5,580,000
bAST3	428,550	103,830(24%)	324,720(75%)	2,670	28,200	431,220	87,570(20%)	87,570(20%)	13,795,440	13,212,720
bN1	150	0(0%)	150(100%)	0	0	150	0(0%)	0(0%)	3,000	3,000
bN2	180	150(83%)	30(16%)	60	9,000	240	120(66%)	120(66%)	22,440	7,680
bN3	240	240(100%)	0(0%)	60	9,000	300	180(75%)	180(75%)	22,680	7,560
bPP1	90,600	46,200(50%)	44,400(49%)	5,600	436,800	96,200	46,200(50%)	46,200(50%)	4,214,400	3,033,600
bPP2	78,000	44,800(57%)	33,200(42%)	6,600	476,800	84,600	44,800(57%)	44,800(57%)	3,790,400	2,571,200
bPP3	546,710	398,420(72%)	148,290(27%)	52,860	6,475,120	599,570	398,420(72%)	398,420(72%)	29,103,720	17,192,120
bReg1	1,000	200(20%)	800(80%)	0	0	1,000	100(10%)	100(10%)	34,400	33,600
bReg2	2,162,830	427,970(19%)	1,734,860(80%)	427,950	17,118,080	2,162,860	10(0%)	10(0%)	86,513,920	84,799,800
bReg3	1,949,950	476,010(24%)	1,473,940(75%)	476,020	19,042,680	1,950,010	10(0%)	10(0%)	78,001,720	76,093,720
bR1	400,011	7(0%)	400,004(99%)	46	2,631,600	400,055	0(0%)	3(0%)	17,263,480	13,023,236
bR2	2,530	1,583(62%)	947(37%)	117	15,608	2,642	289(11%)	299(11%)	141,056	99,404
bR3	79,456	53,259(67%)	26,197(32%)	4,809	686,196	84,073	13,365(16%)	13,454(16%)	7,701,916	6,045,808
total	**6,129,207**	**1,637,669(26%)**	**4,491,538(73%)**	**980,792**	**46,953,084**	**6,205,920**	**670,064(10%)**	**670,166(10%)**	**253,288,572**	**228,171,448**
Total	**8,605,147**	**1,980,005(23%)**	**6,625,142(76%)**	**1,002,212**	**62,476,208**	**8,701,783**	**874,802(10%)**	**876,043(10%)**	**342,818,892**	**296,863,696**

Table 3: Original benchmark (baseline for all the other measurements)

bench.	*NC*	*NNEC*	*NEC*	*NCE*	*NCB*	*NAC*	*NOSM*	*NSM*	*NAB*	*NUB*
1	643,603	151,665(23%)	491,938(76%)	2,636	35,940	229,379	112,308(17%)	112,439(17%)	3,956,792	3,044,076
2	13	12(92%)	1(7%)	0	0	13	1(7%)	1(7%)	468	392
3	79,035	34,970(44%)	44,065(55%)	45	2,272	44,294	10,612(13%)	10,689(13%)	1,397,668	1,236,260
4	1,690	512(30%)	1,178(69%)	30	1,720	905	245(14%)	245(14%)	33,160	22,820
5	3,551	2,009(56%)	1,542(43%)	30	1,208	2,685	200(5%)	251(7%)	65,208	55,124
6	96	44(45%)	52(54%)	0	0	91	40(41%)	40(41%)	3,560	3,284
7	644	218(33%)	426(66%)	614	2,257,960	927	48(7%)	48(7%)	4,509,216	689,468
8	2	0(0%)	2(100%)	0	0	0	0(0%)	0(0%)	0	0
9	158,395	58,331(36%)	100,064(63%)	5,663	280,876	123,703	57,637(36%)	57,718(36%)	5,220,152	3,921,204
10	1,432,306	14,891(1%)	1,417,415(98%)	6,074	12,258,424	68,132	8,248(0%)	8,344(0%)	19,547,672	6,830,248
11	6,839	2,058(30%)	4,781(69%)	1,280	78,712	2,186	471(6%)	480(7%)	109,212	75,012
12	7,870	3,472(44%)	4,398(55%)	1,094	44,384	5,253	20(0%)	20(0%)	194,732	84,716
13	108,571	57,590(53%)	50,981(46%)	1,739	369,336	62,510	8,151(7%)	8,810(8%)	4,218,656	3,286,404
14	10,305	586(5%)	9,719(94%)	145	14,044	963	82(0%)	110(1%)	136,008	112,532
15	20,815	14,886(71%)	5,929(28%)	255	125,900	18,489	5,736(27%)	5,740(27%)	2,640,784	1,932,632
16	766	172(22%)	594(77%)	17	496	221	123(16%)	123(16%)	14,288	10,452
17	1,203	880(73%)	323(26%)	1,512	48,384	2,392	764(63%)	764(63%)	120,856	29,488
total	**2,475,704**	**342,296(13%)**	**2,133,408(86%)**	**21,134**	**15,519,656**	**562,143**	**204,686(8%)**	**205,822(8%)**	**42,168,432**	**21,334,112**
bAST1	210,000	38,000(18%)	172,000(81%)	0	0	47,000	38,000(18%)	38,000(18%)	820,000	536,000
bAST2	179,000	47,000(26%)	132,000(73%)	4,000	24,000	53,000	41,000(22%)	41,000(22%)	1,016,000	668,000
bAST3	428,550	103,830(24%)	324,720(75%)	2,670	28,200	113,040	87,570(20%)	87,570(20%)	2,389,680	1,806,960
bN1	150	0(0%)	150(100%)	0	0	0	0(0%)	0(0%)	0	0
bN2	180	150(83%)	30(16%)	60	9,000	210	120(66%)	120(66%)	21,840	7,080
bN3	240	240(100%)	0(0%)	60	9,000	300	180(75%)	180(75%)	22,680	7,560
bPP1	90,600	46,200(50%)	44,400(49%)	5,600	436,800	78,000	46,200(50%)	46,200(50%)	3,490,400	2,309,600
bPP2	78,000	44,800(57%)	33,200(42%)	6,600	476,800	70,200	44,800(57%)	44,800(57%)	3,218,400	1,999,200
bPP3	546,710	398,420(72%)	148,290(27%)	52,860	6,475,120	543,770	398,420(72%)	398,420(72%)	26,952,320	15,040,720
bReg1	1,000	200(20%)	800(80%)	0	0	200	100(10%)	100(10%)	7,200	6,400
bReg2	2,162,830	427,970(19%)	1,734,860(80%)	427,950	17,118,080	428,000	10(0%)	10(0%)	17,120,000	15,405,880
bReg3	1,949,950	476,010(24%)	1,473,940(75%)	476,020	19,042,680	476,070	10(0%)	10(0%)	19,044,600	17,136,600
bR1	400,011	7(0%)	400,004(99%)	46	2,631,600	52	0(0%)	3(0%)	5,263,360	1,223,116
bR2	2,422	1,583(65%)	839(34%)	117	15,608	1,698	289(11%)	299(12%)	109,356	67,704
bR3	78,145	53,259(68%)	24,886(31%)	4,809	686,196	63,400	13,365(17%)	13,454(17%)	7,034,296	5,378,188
total	**6,127,788**	**1,637,669(26%)**	**4,490,119(73%)**	**980,792**	**46,953,084**	**1,874,940**	**670,064(10%)**	**670,166(10%)**	**86,510,132**	**61,393,008**
Total	**8,603,492**	**1,979,965(23%)**	**6,623,527(76%)**	**1,001,926**	**62,472,740**	**2,437,083**	**874,750(10%)**	**875,988(10%)**	**128,678,564**	**82,727,120**

Table 4: Lazy internal array creation

bench.	*NC*	*NNEC*	*NEC*	*NCE*	*NCB*	*NAC*	*NOSM*	*NSM*	*NAB*	*NUB*
1	643,603	151,665(23%)	491,938(76%)	2,636	35,940	226,736	112,308(17%)	112,439(17%)	3,921,136	3,044,076
2	13	12(92%)	1(7%)	0	0	13	1(7%)	1(7%)	468	392
3	78,977	34,912(44%)	44,065(55%)	45	2,272	44,193	10,604(13%)	10,681(13%)	1,393,396	1,234,432
4	1,690	512(30%)	1,178(69%)	30	1,720	873	245(14%)	245(14%)	31,200	22,820
5	3,551	2,009(56%)	1,542(43%)	30	1,208	2,683	200(5%)	251(7%)	65,168	55,124
6	96	44(45%)	52(54%)	0	0	91	40(41%)	40(41%)	3,560	3,284
7	644	218(33%)	426(66%)	614	2,257,960	235	48(7%)	48(7%)	2,372,616	689,468
8	2	0(0%)	2(100%)	0	0	0	0(0%)	0(0%)	0	0
9	158,399	58,333(36%)	100,066(63%)	5,663	280,876	118,044	57,639(36%)	57,720(36%)	4,938,224	3,921,356
10	1,432,306	14,891(1%)	1,417,415(98%)	6,074	12,258,424	62,065	8,248(0%)	8,344(0%)	11,877,480	6,830,244
11	6,839	2,058(30%)	4,781(69%)	1,280	78,712	2,056	471(6%)	480(7%)	91,804	75,012
12	7,870	3,472(44%)	4,398(55%)	1,094	44,384	4,163	20(0%)	20(0%)	144,144	84,716
13	108,571	57,590(53%)	50,981(46%)	1,739	369,336	61,326	8,151(7%)	8,810(8%)	3,907,184	3,286,404
14	10,305	586(5%)	9,719(94%)	145	14,044	817	82(0%)	110(1%)	119,512	112,532
15	20,815	14,886(71%)	5,929(28%)	255	125,900	18,233	5,736(27%)	5,740(27%)	2,514,840	1,932,620
16	766	172(22%)	594(77%)	17	496	203	123(16%)	123(16%)	13,700	10,452
17	1,203	880(73%)	323(26%)	1,512	48,384	882	764(63%)	764(63%)	72,536	29,488
total	**2,475,650**	**342,240(13%)**	**2,133,410(86%)**	**21,134**	**15,519,656**	**542,613**	**204,680(8%)**	**205,816(8%)**	**31,466,968**	**21,332,420**
bAST1	210,000	38,000(18%)	172,000(81%)	0	0	47,000	38,000(18%)	38,000(18%)	820,000	536,000
bAST2	179,000	47,000(26%)	132,000(73%)	4,000	24,000	49,002	41,000(22%)	41,000(22%)	992,012	668,000
bAST3	428,550	103,830(24%)	324,720(75%)	2,670	28,200	110,370	87,570(20%)	87,570(20%)	2,361,480	1,806,960
bN1	150	0(0%)	150(100%)	0	0	0	0(0%)	0(0%)	0	0
bN2	180	150(83%)	30(16%)	60	9,000	153	120(66%)	120(66%)	13,400	7,080
bN3	240	240(100%)	0(0%)	60	9,000	243	180(75%)	180(75%)	14,240	7,560
bPP1	91,000	46,400(50%)	44,600(49%)	5,600	437,600	72,603	46,400(50%)	46,400(50%)	3,058,196	2,312,800
bPP2	78,000	44,800(57%)	33,200(42%)	6,600	476,800	63,604	44,800(57%)	44,800(57%)	2,743,088	2,000,000
bPP3	546,710	398,420(72%)	148,290(27%)	52,170	6,449,480	490,915	398,420(72%)	398,420(72%)	20,488,808	15,051,560
bReg1	1,000	200(20%)	800(80%)	0	0	200	100(10%)	100(10%)	7,200	6,400
bReg2	2,162,830	427,970(19%)	1,734,860(80%)	427,950	17,118,080	427,970	10(0%)	10(0%)	17,119,200	15,405,880
bReg3	1,949,950	476,010(24%)	1,473,940(75%)	476,020	19,042,720	476,011	10(0%)	10(0%)	19,042,492	17,136,640
bR1	400,011	7(0%)	400,004(99%)	46	2,631,600	38	0(0%)	3(0%)	5,243,040	1,023,116
bR2	2,422	1,583(65%)	839(34%)	117	15,608	1,597	289(11%)	299(12%)	94,656	67,712
bR3	78,145	53,259(68%)	24,886(31%)	4,872	699,036	58,872	13,365(17%)	13,454(17%)	6,375,776	5,390,776
total	**6,128,188**	**1,637,869(26%)**	**4,490,319(73%)**	**980,165**	**46,941,124**	**1,798,578**	**670,264(10%)**	**670,366(10%)**	**78,373,588**	**61,420,484**
Total	**8,603,838**	**1,980,109(13%)**	**6,623,729(86%)**	**1,001,299**	**62,460,780**	**2,341,191**	**874,944(8%)**	**876,182(8%)**	**109,840,556**	**82,752,904**

Table 5: Lazy internal array creation + reuse of array

bench.	*NC*	*NNEC*	*NEC*	*NCE*	*NCB*	*NAC*	*NOSM*	*NSM*	*NAB*	*NUB*
1	643,626	151,687(23%)	491,939(76%)	2,648	35,956	226,964	112,329(17%)	112,460(17%)	3,906,608	3,029,456
2	13	12(92%)	1(7%)	0	0	13	1(7%)	1(7%)	468	392
3	79,043	34,974(44%)	44,069(55%)	65	2,108	44,259	10,616(13%)	10,693(13%)	1,384,428	1,225,288
4	1,691	513(30%)	1,178(69%)	30	1,760	874	246(14%)	246(14%)	30,952	22,532
5	3,551	2,009(56%)	1,542(43%)	28	1,120	2,683	200(5%)	251(7%)	65,352	55,308
6	96	44(45%)	52(54%)	0	0	91	40(41%)	40(41%)	3,560	3,284
7	644	218(33%)	426(66%)	614	2,257,960	235	48(7%)	48(7%)	2,372,608	689,492
8	2	0(0%)	2(100%)	0	0	0	0(0%)	0(0%)	0	0
9	158,399	58,333(36%)	100,066(63%)	5,651	280,636	118,043	57,639(36%)	57,720(36%)	4,938,340	3,921,516
10	1,432,306	14,891(1%)	1,417,415(98%)	6,069	12,280,124	62,070	8,248(0%)	8,344(0%)	11,852,428	6,828,504
11	6,839	2,058(30%)	4,781(69%)	1,280	78,760	2,055	471(6%)	480(7%)	91,812	75,064
12	7,870	3,472(44%)	4,398(55%)	1,095	46,236	4,163	20(0%)	20(0%)	145,360	86,528
13	108,571	57,589(53%)	50,982(46%)	1,683	367,164	61,330	8,150(7%)	8,809(8%)	3,865,908	3,245,964
14	10,305	586(5%)	9,719(94%)	145	14,044	819	82(0%)	110(1%)	120,872	112,660
15	20,815	14,886(71%)	5,929(28%)	255	125,900	18,233	5,736(27%)	5,740(27%)	2,514,084	1,932,656
16	766	172(22%)	594(77%)	14	376	202	123(16%)	123(16%)	12,488	9,428
17	1,203	880(73%)	323(26%)	0	0	880	764(63%)	764(63%)	72,472	29,488
total	**2,475,740**	**342,324(13%)**	**2,133,416(86%)**	**19,577**	**15,492,144**	**542,914**	**204,713(8%)**	**205,849(8%)**	**31,377,740**	**21,267,560**
bAST1	210,000	38,000(18%)	172,000(81%)	0	0	47,000	38,000(18%)	38,000(18%)	820,000	536,000
bAST2	179,000	47,000(26%)	132,000(73%)	4,000	24,000	49,002	41,000(22%)	41,000(22%)	992,012	668,000
bAST3	428,550	103,830(24%)	324,720(75%)	2,670	28,200	110,370	87,570(20%)	87,570(20%)	2,329,080	1,774,560
bN1	150	0(0%)	150(100%)	0	0	0	0(0%)	0(0%)	0	0
bN2	180	150(83%)	30(16%)	60	9,000	154	120(66%)	120(66%)	11,280	4,920
bN3	240	240(100%)	0(0%)	60	9,000	244	180(75%)	180(75%)	12,120	5,400
bPP1	90,600	46,200(50%)	44,400(49%)	5,600	437,600	72,403	46,200(50%)	46,200(50%)	3,057,396	2,312,800
bPP2	78,000	44,800(57%)	33,200(42%)	6,600	476,800	63,604	44,800(57%)	44,800(57%)	2,743,088	2,000,000
bPP3	546,710	398,420(72%)	148,290(27%)	52,170	6,449,480	490,915	398,420(72%)	398,420(72%)	20,488,808	15,051,560
bReg1	1,000	200(20%)	800(80%)	0	0	200	100(10%)	100(10%)	7,200	6,400
bReg2	2,162,830	427,970(19%)	1,734,860(80%)	427,950	17,118,080	427,970	10(0%)	10(0%)	17,119,200	15,405,880
bReg3	1,949,950	476,010(24%)	1,473,940(75%)	476,020	19,042,720	476,011	10(0%)	10(0%)	19,042,492	17,136,640
bR1	400,011	7(0%)	400,004(99%)	46	2,631,600	38	0(0%)	3(0%)	5,243,040	1,023,116
bR2	2,422	1,583(65%)	839(34%)	117	15,608	1,597	289(11%)	299(12%)	94,616	67,672
bR3	78,145	53,259(68%)	24,886(31%)	4,872	699,036	58,872	13,365(17%)	13,454(17%)	6,375,736	5,390,736
total	**6,127,788**	**1,637,669(26%)**	**4,490,119(73%)**	**980,165**	**46,941,124**	**1,798,380**	**670,064(10%)**	**670,166(10%)**	**78,336,068**	**61,383,684**
Total	**8,603,528**	**1,979,993(23%)**	**6,623,535(76%)**	**999,742**	**62,433,268**	**2,341,294**	**874,777(10%)**	**876,015(10%)**	**109,713,808**	**82,651,244**

Table 6: Lazy internal array creation + reuse of array + code refactoring

Reviving Smalltalk-78

The First Modern Smalltalk Lives Again

Dan Ingalls
CDG Labs,
San Francisco, CA, USA
dan@cdglabs.org

Bert Freudenberg
CDG Labs,
Potsdam, Germany
bert@cdglabs.org

Ted Kaehler Yoshiki Ohshima
Alan Kay
Viewpoints Research Institute,
Los Angeles, CA, USA
{alan.kay,ted,yoshiki}@vpri.org

Abstract

We report on a revival of Smalltalk-78 in JavaScript that runs in any web browser. Smalltalk-78 was a port of Smalltalk-76 to the NoteTaker, a portable computer based on the Intel 8086 processor. This same interpreter design and snapshot is the ancestor of Smalltalk-80 and Squeak systems of today. We describe our conversion to a completely different object memory with essentially no visible changes in the language or system, as well as our support of Smalltalk-78's linearized contexts that used the 8086 stack directly. We report how the Lively Web development system facilitated tooling for the project, as well as the integration of the final result with file access over the Internet. We report several performance results and describe how the resurrected system and IDE was actually used to build an entire slide composition and presentation system used to produce a present-day illustrated talk.

Figure 1. NoteTaker hardware

1. Background

Smalltalk-76 was the first modern Smalltalk, combining a compilable keyword message syntax with a compact and efficient byte code interpreter (Ingalls 1978). From its precursor (Smalltalk-74) it inherited an object-oriented virtual memory (OOZE) which enabled it to address over a megabyte of objects with sixteen-bit pointers, and a library of line-drawing, bitmap manipulation and text display routines.

At roughly the same time, Intel was developing its 8086 microprocessor to be introduced in 1978, and the Xerox Smalltalk team (see acknowledgements) determined to build a portable computer dubbed "NoteTaker" (Figure 1) around that chip, running Smalltalk as its operating system and application environment.

The challenge of porting Smalltalk-76 to a microprocessor with only 256k bytes of memory led to several more innovations that shaped the future of Smalltalk. The need for efficiency led to an experimental mapping of Smalltalk contexts directly onto the 8086 stack. The overwhelming task of rewriting all the graphics routines from ALTO assembly code to 8086 assembly code motivated a rewrite of all the Smalltalk graphics to use only the single BitBlt primitive operation. By this time the Xerox team had learned enough about how to build an effective IDE, so this port was also an excuse to pare down the system to a fairly lean self-supporting kernel of 100 classes and 2000 methods in an image of 200k bytes.

The NoteTaker Smalltalk, also referred to as Smalltalk-78 (Krasner 1983) is thus a manifestation of the original Smalltalk-76 language and interpreter architecture, together with the hallmark Smalltalk IDE and the earliest BitBlt-based graphic system. Because of its freedom from the complex OOZE virtual memory and the equally complex support for lines, text, and image manipulation in the original Smalltalk-76 system, we determined to revive Smalltalk-78 as a living artifact accessible in any browser.

While only a few NoteTakers were ever built, we are fortunate to have one Smalltalk-78 memory image preserved from that day. It was created by cloning (parts of) the running Smalltalk-76 system. Since this represents a fairly lean snapshot of the original Smalltalk-76 system, we thought it would be a particularly interesting target for reimplementation. A similar cloning process was used to produce an image to run on Xerox's high-performance Dorado machine, and that in turn was stepwise morphed into the publicly released Smalltalk-80 system.[1]

2. Notable features of Smalltalk-78

The constraints on NoteTaker deployment led to a particularly interesting point in the evolution of Smalltalk that can be summarized by the following relationship to its parent:

Language and bytecode architecture Unchanged from the original Smalltalk-76 design (Ingalls 1978).

Dynamic execution model Retains full context semantics, but with a linear stack matched to 8086.

Object memory Complex OOZE virtual memory replaced by simple object table model.

[1] Remnants of this duality can still be seen in the NoteTaker image: lower-level code in various places tests whether it is running on the NoteTaker or Dorado.

Graphics system Large library of assembly code for text, lines, etc. replaced by BitBlt alone.

Programming environment Includes full rich text editor, compiler, paned browser and debugger.

Source code Decompilation used in place of remote source code file.

System size VM reduced from approx 16k in Smalltalk-76 to 6k bytes of 8086 code. Entire IDE was 100 classes with 2000 methods, totalling 200k bytes

3. Revival process

To revive Smalltalk-78 and make it usable in a browser, we determined to leverage the similar SqueakJS project that runs Squeak Smalltalk using n JavaScript in a browser (Freudenberg 2013). We would reuse the innovative object model as is, modify the interpreter and BitBlt for Smalltalk-78 differences, and completely rewrite the context mechanism to follow Smalltalk-78's linear stack model. The SqueakJS browser port was developed in the Lively Web development environment[2], and we took advantage of this support in our revival of Smalltalk-78 as well. Figure 2 shows the initial Notetaker screen.

3.1 The initial snapshot

The Smalltalk-78 snapshot we used was created from a running Smalltalk-76 system. To fit within the NoteTaker hardware, many features were removed from that system.

The snapshot came in two files: an object table dump (30,976 Bytes) and an object memory snapshot (172,592 Bytes). These files were not a direct snapshot, but the result of a conversion program in a running Smalltalk-76 system. These snapshot files were added to the Lively interpreter object as literal arrays. The object table is a sequence of 4-byte entries. Each entry encodes the data address, along with some other bits including a reference count. Our implementation ignores the reference count since it uses a generational garbage collector.

The object data space is a sequence of 2-byte words. One header word encodes the class oop in the upper 10 bits, and the instance size in the lower 6 bits (class oops always had the lower 6 bits equal to zero in OOZE). If the size field is zero, then there is a word before the class with a 16-bit length. The size field is the object size in bytes, including the class (and size), so a String of length 1 has size = 3, and a Point would have size = 6.

The format of classes is (quoting from the system itself . . .)

- title "<String> for identification, printing"
- myinstvars "<String> partnames for compiling, printing"
- instsize "<Integer> for storage management"
- messagedict "<MessageDict> for communication, compiling"
- classvars "<Dictionary/nil> compiler checks here"
- superclass "<Class> for execution of inherited behavior"
- environment "<Vector of SymbolTables> for external refs"

The instsize is an integer (i.e. low bit = 1) with the following interpretation:

- 0x8000 – fields are oops, else not
- 0x4000 – fields are words, else bytes
- 0x2000 – instances are variable length
- 0x0FFE – instance size in words including class

[2] See `http://lively-web.org/`

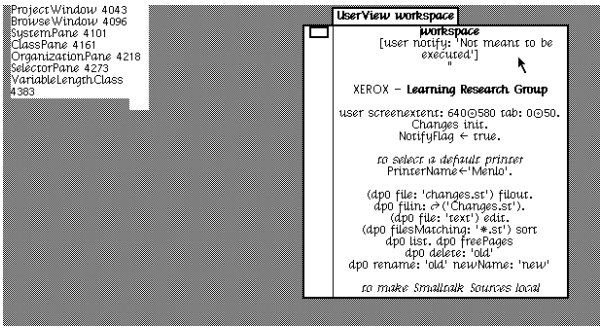


Figure 2. The initial NoteTaker screen.[3]

Thus Point has instsize = 0x8006 (class + 2 pointers) and Float has instsize = 0x4008 (class + 3 words). Floats on the NoteTaker use a non-standard format: 15 bits exponent in two's complement, 1 bit sign, and 32 bits mantissa. These floats are converted to standard IEEE 754 doubles (1 sign bit, 11 bits biased exponent, 53 bits mantissa) by bitshifts, resulting in native JavaScript numbers.

We discovered that some objects in the snapshot did not have the right class set. Specifically, we noticed that e.g. Array new: 5 failed with an inexplicable error. It turned out that the Array class was supposed to be an instance of VariableLengthClass which implements new:. But in the snapshot, it was an instance of Class. This problem did not manifest in the original NoteTaker VM because it intercepted new: as a primitive without actually looking it up. All other variable-length classes (String, UniqueString, Vector, Process, Natural, and CompiledMethod) had the same problem, so this probably came from a bug in the image cloning process.

3.2 The object model

Our VM mimics the Smalltalk-78 bytecode interpreter as faithfully as possible, but has an entirely different storage model from the original, borrowed from SqueakJS (Freudenberg 2013), which in turn was inspired by the Potato system, an implementation of the Squeak VM in Java (Ingalls 2008). Rather than emulating the original object layout scheme with an object table and a contiguous memory area for objects, it maps each Smalltalk object to a JavaScript object, and object references are simply fields of JavaScript objects. Tagged integers are mapped to JavaScript numbers, with typeof checks in places where the VM needs to distinguish objects from integers and floats.

The NoteTaker broke away from OOZE's object zones that placed uncomfortable limits on object size. Our VM goes further still by eliminating the complexities of reference counting. It allows more and larger objects (32K objects, virtually unlimited in size). It could easily be extended to allow even more objects (by representing oops with more than 16 bits in saved snapshots).

Instead of reference counting our VM uses a generational garbage collector as introduced by SqueakJS. Old objects are held in a linked list. New objects are not explicitly referenced, enabling them to be garbage-collected by the host GC of JavaScript. An object gets tenured when its oop is needed for the first time (typically for hashing). A full GC sweeping old and new space is rare, basically only for become operations and when snapshotting. This makes garbage disposal very efficient, since the vast majority of it is not handled explicitly but by the host language.

[3] The live system is available at `http://lively-web.org/users/bert/Smalltalk-78.html` (Freudenberg and Ingalls 2014). The initial snapshot can be loaded as "original image".

```
FIRST_TEMP: -1,// temps, followed by callee's stack
SAVED_BP: 0,   // rel link to caller's frame
CALLER_PC: 1,  // caller's supended PC
NUMARGS: 2,    // args were stacked right to left
METHOD: 3,     // method
MCLASS: 4,     // method class (needed for super)
RECEIVER: 5,   // top stack item in caller's frame
LAST_ARG: 6,   // stack item in caller's frame
```

Figure 3. Context frame layout (relative to BP)

```
MINSIZE: 0,     //
HWM: 1,         // not used
TOP: 2,         // top of stack rel to end
RESTARTCODE: 3, // code to run for restart
STACK: 4,       // bottom of stack ares
```

Figure 4. Process layout

3.3 Linear stacks

It was enough of a stretch to make a bytecode interpreter for Smalltalk perform acceptably on any machines available in 1978, but getting it to run on a 4MHz 8086 microcomputer required the Xerox team to explore every possible trick to improve speed. Because the 8086 offered special instructions for pushing and popping values on its stack, and also for switching from one frame of temporary variables to another, Smalltalk-76's general Context objects were recast into a series of overlapping context-like frames in the 8086's linear stack.

Figure 3 shows the layout of a Smalltalk-78 stack frame. The base pointer (BP) pointed to the base of the context frame, and this location held a link to the caller's base pointer, followed by the caller's supended PC. The remaining locations were similar to a normal Smalltalk context. Note that the stack grows downward toward lower addresses: arguments are first pushed, then the receiver, then the method info. This is followed by the caller's PC, and a link to the caller's BP, after which comes the state of the new context with temps first followed by the new context's stack cells. A conscious aspect of this design is that the layout of the top of the caller's frame is the same as the base of of the called frame, and these can simply be aliased so that no copying of receiver or arguments, nor allocation of a new context object is required to perform a normal Smalltalk message send.

The linear stacks themselves were Smalltalk objects of class Process, and these could be activated alernately to achieve a process switch. Each process object included a header that stored its current top of stack location, and it was possible to grow and shrink these variable-length objects as needed. Figure 4 shows the layout of a process object with many context frames in it, and Figure 6 shows an example.

The alert reader will recognize that it is not possible to emulate the operation of Smalltalk blocks with a linear stack alone. Smalltalk-78's RemoteCode objects (shown in Figure 5) provided the necessary added PC and remote return point needed to support out-of-line block execution, analogous to the RemoteContexts of Smalltalk-76. While use of remote code was no faster than in Smalltalk-76, almost all sends (well over 99 percent) in typical code ran using the overlapping stack frames.

3.4 BitBlt display

Smalltalk-78's BitBlt is relatively simple, supporting only black-and-white bitmaps, a 4x4 halftone pattern, four source rules (src, not src, halftone in src, halftone), and four combination rules (store, or, xor, and). Since each word stores 16 pixels, operations are relatively fast. For even more performance we use specialized inner loops, for example for filling, and for copying with the store rule.

To display the bits on the screen we use an HTML canvas. We create a JavaScript ImageData object, which can be displayed on the canvas in a single drawing call. It needs 32-bit RGBA data, which we create pixel-by-pixel from the bits in the display bitmap. Doing this for the full screen (1024x768 by default) would still be quite expensive. Instead, we only do it for the rectangle affected by each BitBlt operation. Moreover, we record these "dirty" rectangles and merge them if possible, only actually flushing to the canvas when needed. This has the nice effect of reducing flicker, since not every individual drawing operation is seen by the user.

```
FRAMEOFFSET: 0, // offset of my frame in process
STARTINGPC: 1,  // PC to start or restart
PROCESS: 2,     // my process
STACKOFFSET: 3, // my saved stack pointer
```

Figure 5. RemoteCode layout

```
[1970]  savedBP: 8
[1971] callerPC: 139
[1972]  numArgs: 0
[1973]   method: a CompiledMethod: Window>>eachtime
[1974]   mclass: the Window class
[1975] receiver: a BitRectEditor
[1976] temp3/t4: true
[1977] temp2/t3: a BitRectEditor
[1978] temp1/t2: 1
[1979]  savedBP: 6
[1980] callerPC: 9
[1981]  numArgs: 1
[1982]   method: a CompiledMethod: UserView>>run:
[1983]   mclass: the UserView class
[1984] receiver: an UserView
[1985]  arg0/t1: false
[1986]  savedBP: 5
[1987] callerPC: 31
[1988]  numArgs: 0
[1989]   method: a CompiledMethod: UserView>>run
[1990]   mclass: the UserView class
[1991] receiver: an UserView
[1992]  savedBP: 5
[1993] callerPC: 103
[1994]  numArgs: 0
[1995]   method: a CompiledMethod: Process>>run
[1996]   mclass: the Process class
[1997] receiver: a Process
[1998]  savedBP: 0
[1999] callerPC: 0
[2000]  numArgs: 0
[2001]   method: a CompiledMethod: Process>>goBaby
[2002]   mclass: the Process class
[2003] receiver: a Process
```

Figure 6. Example stack frames, as shown in the VM debugger. The method Process≫goBaby was used to bootstrap the system, it is only executed when starting up the original image, but remains the top stack frame. The receiver slot always overlaps between frames, and if there were arguments, those too (e.g. at index 1984/1985: false and self were pushed by UserView≫run, then run: was sent)

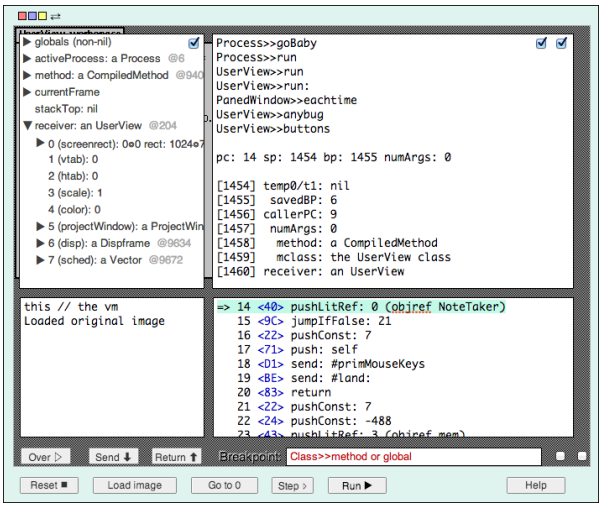


Figure 7. The debugger interface. It shows an object inspector in the upper left, an eval pane below it, the current stack in the upper right, the bytecodes of the current method in the lower right.

It was somewhat tricky to get the flushing right though: the original system did not use double-buffering[4]. We flush whenever an input primitive is called, because then we can assume that all intermediate drawing operations are finished. This works very well in general, but not for animations without user input checks. E.g. a "flash" operation reverses a portion of the screen twice. If we flush the screen after that, no change would be visible. So we had to modify the Smalltalk code by inserting a flush between the two reversals.

The canvas also shows the 16x16 pixel mouse cursor. We erase it from its previous position by drawing those pixels from the display bitmap just as after a BitBlt. Then it is shown at its new position.

3.5 Lively Debugging facilities

Having each Smalltalk object be a JavaScript object makes this VM convenient to debug using the Lively Web interface. Even before we had develped a nice VM viewer we could use Lively's inspectors and workspaces to interact with the VM. We soon added a bytecode disassembler and stack display to trace the execution, and a hierarchical inspector to explore object trees. These facilities make good use of the reflective nature of Smalltalk: While the VM normally does not care about instance variable names or the contents of selectors, we decoded them to make the debug display more meaningful (Figure 7).

Another helpful feature to get the system going were the decompiled sources. From independent work by Helge Horch, we had a complete set of decompiled source code, one class per file, and we attached to our debugger a little viewer that automatically jumped to the source code of a method when it was invoked. Since this used HTML text we had to map the unusual Smalltalk-78 characters to Unicode characters as best we could. Interestingly, not all the characters needed exist in Unicode[5]. We had to use some non-obvious mappings, like an open triangle for the open colon (Figure 8).

[4] Instead, it relied on specially selected "slow" phosphors in the cathode ray tube to reduce flicker.

[5] Perhaps there should be an effort to make them into official Unicode symbols? APL got its own section with all operators. We would need a white colon in particular, and perhaps an eyeball, quote, prompt and do-it chars

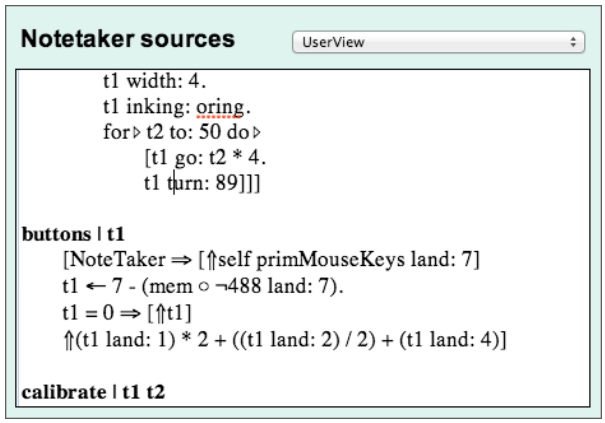


Figure 8. The sources view

4. Using a 36-year-old Smalltalk

4.1 Speed and Space

As the frequency of bugs dwindled, we were surprised how pleasant it was to use this old Smalltalk. This was partly a fortuitous result of the way in which we brought the system back to life. With the web browser came convenience and large clear bitmap graphics; with modern processors came more speed than the original native code; with our new object model most object size restrictions vanished, along with the need for any attention to reference counts.

Once things were running, there was still much work to be done, since Smalltalk-78 was never really finished. It was completed to the point of demonstration on the few NoteTakers that were actually built, but the machines were difficult to use with their small screens and marginal performance, and it was not easy to capture changes and feed them back into new releases. Originally, the Xerox group wrote a Smalltalk-78 image from a running Smalltalk-76 system and this was then moved to the NoteTaker and tried. After several iterations, one image worked well enough for demos. While a number of fixes were made and stored on NoteTaker floppy disks, those are long gone and they were never folded back into the snapshot we have.

4.2 Finishing the job

As our reimplementation became usable (more so than the original), the entire team began working in it as though they had just downloaded a completely modern tool. It was gratifying to see the original design validated in such a way.

We fell almost instinctively into the process of "finishing" this software. This included such tasks as ...

- Making a convenient mechanism for saving and distributing changeSets
- Making an automatic update system for installing newly released changeSets
- Fixing bugs (there were several)
- Removing unused methods
- Taking advantage of the considerable increase in speed. For instance finding all senders of a message had been so slow that it was done by executing a code snippet in a workspace. With the greater speed, it became natural to present such retrievals as menu commands

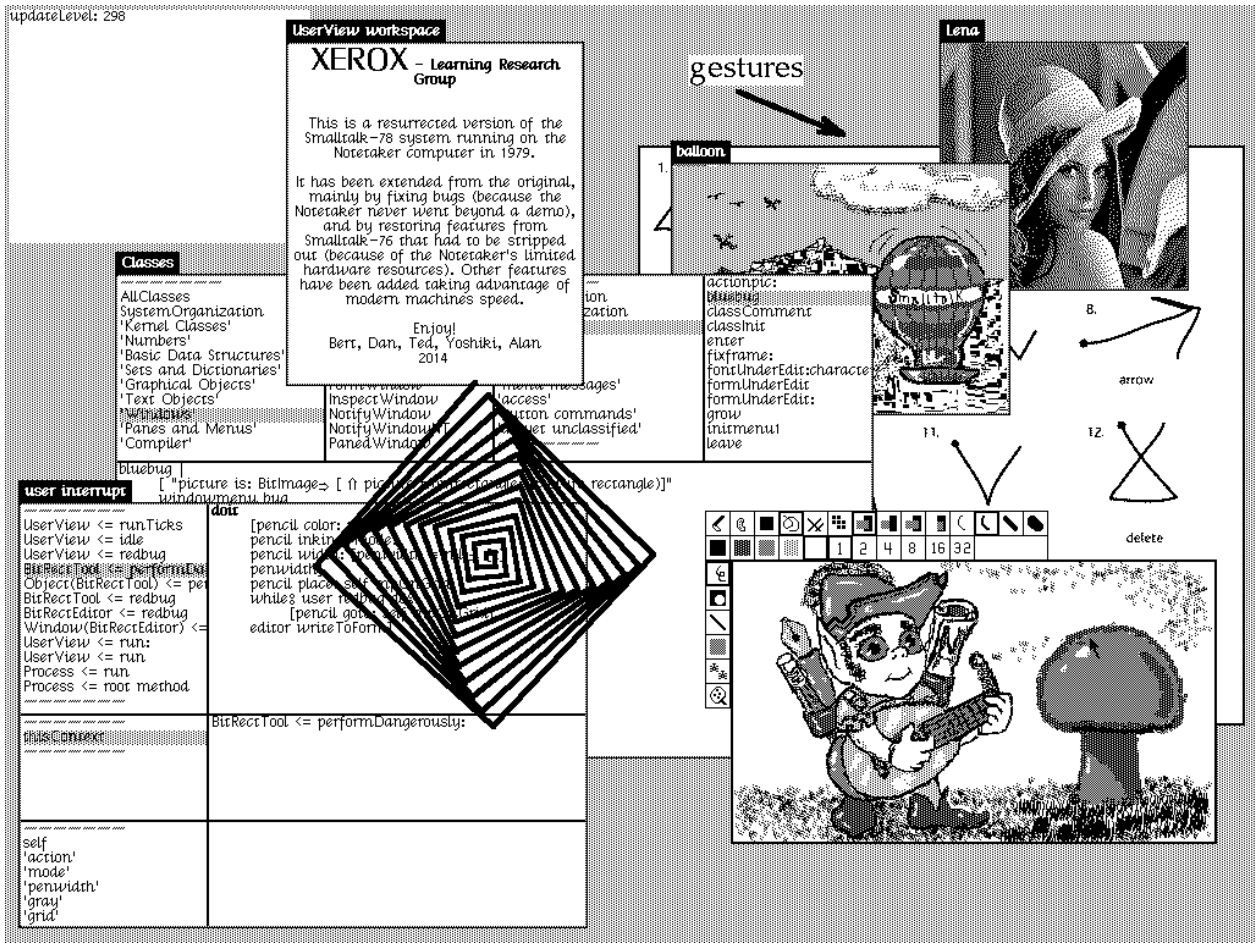


Figure 9. Various tools in the updated Smalltalk-78 system

- Completing the support of the excellent Smalltalk-76 debugger which had never been made to work completely with Smalltalk-78's linearized stack
- Recovering source code from an independently recovered Smalltalk-76 file

4.3 Recovering the source sode

Source code for a method in the NoteTaker had to be decompiled from the bytecodes of the method due to limited memory space. Decompiled code lacks meaningful names for temporary variables and it is also devoid of comments. Since the Smalltalk-78 snapshot was mechanically generated from a Smalltalk-76 most methods are identical to their Smalltalk-76 parents. With plenty of space avilable in the revived Smalltalk-78, we made an effort to restore full source code.

We were fortunate to find one file of source code for Smalltalk-76, although we had no way of knowing how well it matched our Smalltalk-78 snapshot. The file was simply a concatenation of all the methods with no indication of what class they came from, and only separated by arcane markers from a bygone text editor. We managed to isolate the methods and determine their classes in most cases, and then read them with an importer that would only accept methods if they generated the same bytecodes (actually if they produced the same decompilation) as the corresponding methods in our snapshot. (See Appendix B for the details.)

An immediate benefit from decoding the sources file was that we were able to import Kaehler's BitRectEditor, a tool similar to MacPaint, but developed in 1975 in an earlier Smalltalk.[6] The BitRectEditor has programmable tools, each composed of a texture ink, a BitBlt mode, and a nib. When a tool is selected, its components are shown in the top menu. A tool can be reconfigured by clicking and new tools can be created on the fly.

4.4 Life in the Cloud

With Smalltalk-78 running in the browser, work within the system became much more productive, but access to external files for reading and writing was actually more difficult than before. Here we were able to take advantage of hosting in the Lively Web to make access to changes files, snapshots, and image resources actually easier than before.

[6] In the earlier Smalltalks, the whole screen was too large to fit into a single Smalltalk object. Therefore images were stored as BitRects—objects that held striped data in 2k-byte chunks (which was optimal for OOZE). The image painting tool (BitRectEditor) would paint the BitRect's bits on the screen, do the editing using BitBlt on the screen only, and scrape the bits back into stripes in the BitRect when done.

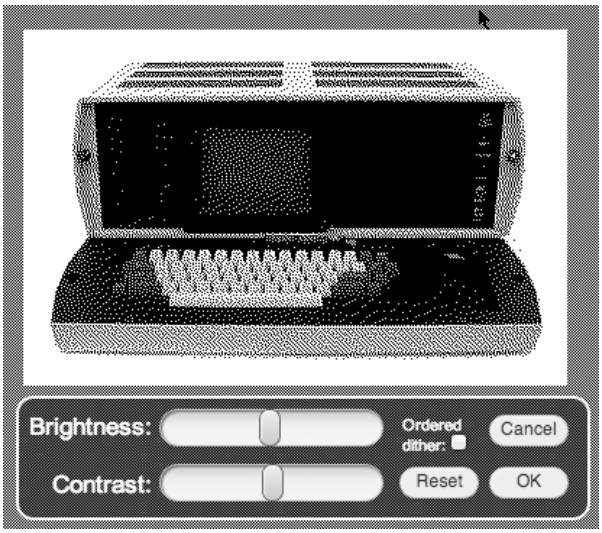


Figure 10. Bitmap importer with dithering UI

Files We reappropriated the existing "port" primitive (which had been used for file i/o) to implement a simple string-based file interface. It takes a file name string and a file contents string and stores them in a JavaScript dictionary. It is also written to the web browser's `localStorage`, which survives reloading of the browser page and thus provides persistence. A list of files is returned when passing an empty file name. These (and more) methods are provided by the user global, the current UserView instance:

```
1  user fileString: 'temp.txt' ← 'foo␣bar␣baz'.   "create file"
2
3  user fileString: 'temp.txt'        "retrieve file"
4          ⇒ 'foo␣bar␣baz'
5
6  user fileString: 'temp.txt' ← nil.      "delete file"
```

Files can be imported by drag-and-drop and are then available to Smalltalk.

Bitmaps A special case is importing bitmap files like JPEGs or PNGs. The NoteTaker code obviously can not load these files directly, as the formats had only been invented decades later. It did, however, define a binary serialization format (asInstance and fromInstance:) for black-and-white forms. This consists simply of the instance variables of the Form, a few Integers followed by a String for the bits. The Integers are stored as 16 bit big-endian numbers, the String has a one or two byte header encoding the length followed by the bytes. To support larger forms we extended this to a four-byte header. When we drop a bitmap into the browser, the system presents a Lively user interface for reducing the color range to black and white. It supports both error diffusion (Floyd/Steinberg algorithm) and ordered dithering. The user can adjust contrast and brightness, then a form file is generated and stored, which can be loaded from inside Smalltalk (Figure 10).

Networking The same VM primitive as for accessing local files is used to store files on the server and retrieve them. If the filename starts with "http:" then instead of storing it locally, we access it via a WebDAV server provided by Lively.[7] If the filename ends in a slash, a list of file names in that directory is returned. This enables seamless working in the cloud or locally.

[7] It is amusing to use the term "http" in a system that predates the invention of HTTP.

Table 1. Interpreter performance, measured on a laptop with a 2.2 GHz Intel Core i7 CPU

	Bytecodes/sec	Sends/sec
Chrome 35.0	1,600,000	70,000
Firefox 30.0	6,900,000	110,000
Safari 7.0.4	9,300,000	350,000

Update stream Using the network file access we set up an update mechanism to distribute changeSets. Each changeSet is a separate file. There is an index file named "updates.list" containing a list of all update file names. The image maintains its own number of loaded updates, so it knows how many new updates need to be loaded from the list. There are now already hundreds of changeSets in that stream. Because loading them from the start takes a long time, we also provide an updated image that is automatically downloaded when a user starts Smalltalk-78 for the first time.

4.5 Performance

We ported the tinyBenchmarks from Squeak to Smalltalk-78 for measuring raw bytecode and send speeds. Results are shown in Table 1. Performance depends considerably on the web browser's JavaScript VM. For this particular workload, Safari's Webkit JIT compiler outperforms Chrome's V8 engine by orders of magnitude, with Firefox in the middle.

In reality we throttle the interpreter when it is idle, as the interpreter speed is entirely sufficient. Idle detection works by measuring how often the image calls the input primitives. When it is rapidly reading the mouse and keyboard without the user providing input, we let the VM sleep for up to 200 ms, or until a user event arrives. Thus when the image is busy with some longer operation it will not check for user input, and thus will not be throttled. As soon as the user types something or moves the mouse, the VM resumes at full speed, and keeps going for at least 500 ms before throttling again.

Note that "full speed" here does not actually mean the interpreter runs continuously. Unlike a regular Squeak VM which has a main-loop that is only exited when the application quits, the Smalltalk-78 vm is callback-based, following the design of SqueakJS. That is, the bytecode interpreter loop runs for a limited time only (typically 20 msecs) and then returns control to the web browser. This is necessary so that the web browser can update the screen, which does not happen while JavaScript is executing. Similarly, events can only be processed while no other JavaScript function is active. Once the interpreter loop finishes, a timer event is scheduled to restart the interpreter loop. When throttling, the timeout is 200 ms. When not throttling, it is 0 ms, meaning to call back into the interpreter as soon as possible.

To get the lowest possible delay between user actions and display response, we not only run the VM at full speed while user events arrive. We also break out of the interpret loop early, as soon as something was drawn to the screen (see section 3.4). Otherwise the interpreter would continue using its current time slot and only then return control to the browser. This would delay updating the screen noticeably, and make the system feel sluggish.

4.6 A real test drive

As an experiment to validate not only the underlying design and development system, but also the ability of the system to support an as yet unanticipated application, we built a fairly capable PowerPoint-like presentation system (all in 1-bit graphics, of course). Two example screenshots appear as Figure 11 and Figure 12. The first of these is performing the role of a slide sorter, with a clever BitBlt scheme

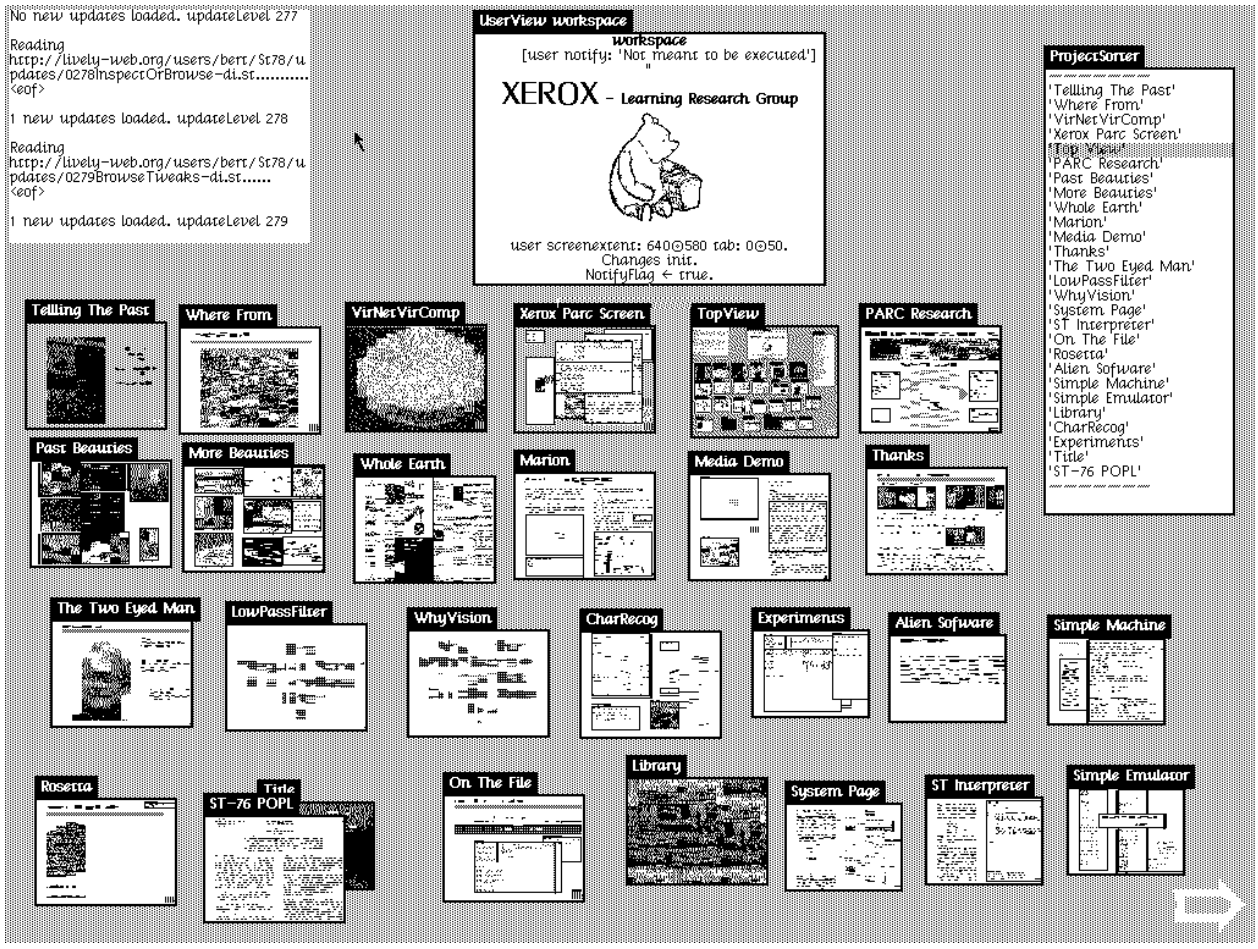


Figure 11. Presentation slides

to produce shrunken thumbnail images on the fly. The second shows one of the slides during a full-screen presentation.

While resurrected software systems are often fragile, we were impressed by the robustness of Smalltalk-78. Our reimplementation had a completely different object memory and a completely different Context discipline, but remained extremely stable throughout our work on "finishing" the system and building this presentation system.

As an example, besides the normal "builds" required in a presentation system, we wanted to support multiple concurrent animations running on the screen while editing and other operations were being done. This required a rework of the window scheduler to provide a queue of ticking objects, and for all existing idle loops to yield to the scheduler in this regard. All of these changes were made in a couple of hours in the running system with very little problem. The picture of the ball in Figure 12 is constantly bouncing (and appearing squashed when it hits the ground) even when other text is being edited.

As mentioned earlier, some interesting features were stripped out in the Smalltalk-78 image from the original Smalltalk-76 image. Most notably, a pen stroke gesture recognizer was missing. To demonstrate the richness of experiments, we have reimplemented a stroke gesture recognizer. The implementation is based on a modern algorithm called $1 recognizer (Wobbrock et al. 2007) rather than the one that was used in the Smalltalk-76 system. The $1 algorithm heavily relies on floating point number computation, but by writing a few primitives to support the algorithm, it works responsively.

5. Things we learned

Small is beautiful. Systems like Smalltalk that are self-describing are highly leveraged. This made it possible to implement Smalltalk-78 in only 6k of 8086 assembly code on the NoteTaker. Similarly we were able to get the system running in a browser with only roughly 3,000 lines of JavaScript. This number grew to 4,000 as we added various comforts such as the support for web-based file access, but the kernel remained small. This same leverage made it a fun project, as we were able to see "bits on the screen" after only about 4 man-weeks of work.

Speed is nice. The improved performance of our implementation over the original made this an exciting project as well. Many facilities that had been barely usable on the NoteTaker and its parent Smalltalk-76 system were delightfully responsive, and the system therefore surprisingly productive.

Clean object API. As with most Smalltalk systems, Smalltalk-78 had a clean interface to storage, and very little work ws needed to completely change from a reference-counted object table model

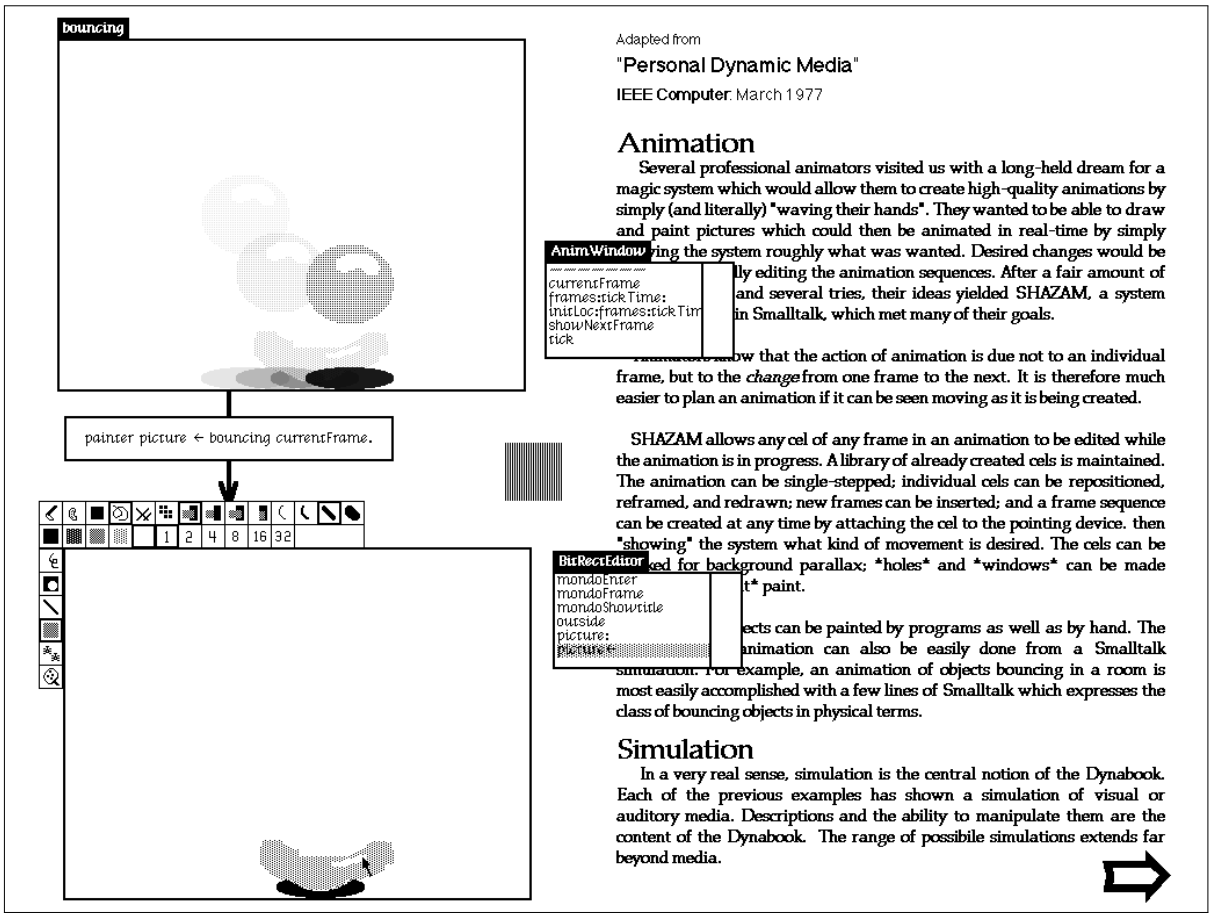


Figure 12. Editing a slide: An animation frame is repainted while the animation is playing. This feature is not "built-in", but was added on-the-fly by connecting two objects with one line of code. (This figure shows a composite of four successive screenshots)

to a direct pointer garbage-collected model. We were, of course, fortunate to inherit a relatively complete JavaScript Smalltalk object model from SqueakJS (Freudenberg 2013).

Browser and Cloud as a universal platform. Finally we learned through this and the earlier SqueakJS project how to adapt the earlier file-based Smalltalk systems to take advantage of the conveniences of browsers and web-based storage facilities. Much of this work was facilitated by our use of the Lively Web development environment, although our completed Smalltalk-78 artifact can operate entirely on its own.

6. Related work

In 2004 Helge Horch got our same Smalltalk-78 snapshot from Dan Ingalls, along with the original 8086 code listings. From this he wrote a relatively complete resurrection in Java, that is yet to be published.

Our Smalltalk-78 VM is based on Bert Freudenberg's "SqueakJS" VM (Freudenberg 2013). It shares the overall design, and parts of the implementation. For example, our BitBlt is a simplification of SqueakJS's BitBlt. SqueakJS in turn was inspired by Dan Ingalls's "Potato", a Squeak VM written in Java (Ingalls 2008).

Another Smalltalk that now runs in a web browser is Smalltalk-72 via Dan Ingalls's Alto emulator (Ingalls 2013). A major difference to our approach is that this emulates Alto machine code which then executes the interpreter, rather than building a new interpreter running the Smalltalk bytecodes.

There are various attempts to implement different languages for the web-browser. Among those, Amber[8] is notable for being a Smalltalk dialect implemented in JavaScript. Most of such languages are implemented as a translator from the language to JavaScript. The key difference in our approach is that we implement a virtual machine that is compatible with the actual old one; this allows us to revive the exact system.

Acknowledgments

The original Smalltalk-78 implementation was done in 1978 mainly by Dan Ingalls and Ted Kaehler (with BIOS by Bruce Horn) at Xerox PARC, and never written up except in the introduction of "Smalltalk-80: Bits of History, Words of Advice" (Krasner 1983, p. 17). A decade later, "The Early History of Smalltalk" (Kay 1993) gave an overview of early Smalltalks, including the NoteTaker version.

The NoteTaker hardware was designed and built by Doug Fairbairn, with help from the hardware support group at PARC. Engineers at Xerox El Segundo debugged the boards. Some NoteTaker

[8] `http://amber-lang.net/`

firmware, schematics, and memos are at `http://bitsavers.org/pdf/xerox/notetaker`

The Smalltalk font in the screenshots is *Cream* by Bob Flegal, based on a design by Alan Kay.

We are grateful to Al Kossow and Dave McDougall for preserving a number of Alto diskpacks, of which the NoteTaker Smalltalk export disk was one. Al and Larry Stewart moved the files to modern media, and they are now well preserved.

References

B. Freudenberg. SqueakJS, 2013. `https://github.com/bertfreudenberg/SqueakJS`.

B. Freudenberg and D. Ingalls. Lively Smalltalk-78, 2014. `http://lively-web.org/users/bert/Smalltalk-78.html`.

D. Ingalls. The Smalltalk-76 programming system design and implementation. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '78, pages 9–16, New York, NY, USA, 1978. ACM. . URL `http://doi.acm.org/10.1145/512760.512762`.

D. Ingalls. JSqueak/Potato, 2008. Original site: `http://web.archive.org/web/20080729033427/http://research.sun.com/projects/JSqueak/`, current home: `https://www.hpi.uni-potsdam.de/hirschfeld/projects/potato/index.html`.

D. Ingalls. Lively Alto emulator running Smalltalk-72, 2013. `http://lively-web.org/users/Dan/ALTO-Smalltalk-72.html`.

A. C. Kay. The early history of Smalltalk. In *The Second ACM SIGPLAN Conference on History of Programming Languages*, HOPL-II, pages 69–95, New York, NY, USA, 1993. ACM. ISBN 0-89791-570-4. . URL `http://doi.acm.org/10.1145/154766.155364`.

G. Krasner. *Smalltalk-80 : bits of history, words of advice*. Addison-Wesley Pub. Co, Reading, Mass, 1983. ISBN 0201116693.

J. O. Wobbrock, A. D. Wilson, and Y. Li. Gestures without libraries, toolkits or training: A $1 recognizer for user interface prototypes. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST '07)*, pages 159–168. ACM, 2007.

A. Smalltalk-78 Syntax

The syntax of Smalltalk-78 is unchanged from Smalltalk-76, which introduced the regularity of unary, binary, and keyword messages that is still used in modern Smalltalks. Readers may refer to the Smalltalk-76 reference for a full description. There are no blocks as in Smalltalk-80, no booleans, and no meta classes.

The system uses the ASCII-1963 character set (which had ↑ and ← arrows in place of the ^ and _ characters in later ASCII versions). From the control character range, only Tab (9) and Carriage Return (13) are used as in ASCII, while some others are used for special characters not found in ASCII. These include return and implication arrows (⇑ ⇒), comparison operators (≤ ≥ ≠ ≡), indexing and point creation operators (∘ ◎), a curved arrow for literal quoting (used like # in Smalltalk-80), a unary minus (for prefixing negative numbers, as in 3 − -4), a possessive operator (evaluates an expression in the context of another object, e.g. obj 's 'instvar'), and a few more. See Figure 13 for a complete list. This code snippet also demonstrates a few idiosyncrasies:

Conditional execution A block of code can be conditionally evaluated using the ⇒ operator. It maps directly to the "jump if not false" bytecode (where false is just an instance of Object known to the VM compared by identity). If the condition is false, the following block of code is skipped. Any object other than false causes the conditional jump not to be taken, so the code block is executed, followed by an unconditional jump to the end of the surrounding code block. This allows putting an "else" case right after the conditional block. If execution should resume after the else case, another set of brackets needs to be put around it, as demonstrated in Figure 13.

Figure 13. Non-ASCII characters used in Smalltalk-78

Deferred evaluation Brackets are merely a syntactic device to group statements, they do not create a block object as in Smalltalk-80. How then does e.g. "for:from:do:" work? The magic is *not* in the brackets, but in the two variants of the colon in a keyword message. The regular colon ":" causes the keyword argument to be evaluated immediately. For an open colon ":" however, the compiler generates a "remote copy" of the current context. This RemoteCode can be evaluated later, and even repeatedly if desired.

There are two variants for evaluation, with and without an argument. The form with an argument (value←) can be used to assign a value to a variable, the form without an argument (eval) just evaluates the code. These work together, as there is no way to pass arguments to a block other than "remotely" assigning to temporary variables. This is perhaps best explained with an example. A possible implementation of "for:from:do:" could look like this:

```
1 for: var from: vec do: code | stream item
2 [
3     stream ← vec asStream.
4     while: [item ← stream next] do: [
5         var value ← item.
6         code eval.
7     ]
8 ]
```

Note that "for:from:do:" is not actually a method, but a compiler macro (as evidenced by the missing receiver). It could, however, be a regular method as just described.

Assignment selector The last part of a keyword selector can be an assignment arrow. E.g. "stream next← 5" writes an item to the stream, whereas "stream next: 5" reads 5 items from the stream. To the method this is just like any other argument. But on the sending side, the parser treats the expression after ← as if it was an assignment. That means no parentheses are needed around that expression. This also works for binary operators extended by an assignment arrow, making it a ternary operator. E.g. "a ∘ 1 ← b ∘ 1" uses the ∘ and ∘ ← operators. It is equivalent to Smalltalk-80's "a at: 1 put: (b at: 1)" but reads a lot nicer and needs no parentheses.

No metaclasses The expression "Stream default" looks like it invokes a "class-side method" if it was Smalltalk-80. But Smalltalk-78 did not have a metaclass hierarchy in parallel to the regular class hierarchy. All classes are an instance of Class so no class-specific methods are possible. Instead, Class provides a couple of methods that dispatch to a new instance. E.g. the implementation of Class≫default is "⇑ self new default". In the Stream case, this invokes Stream≫default which initializes the stream for writing on a new String.

B. Source code recovery

The Smalltalk-76 sources file we found was "Smalltalk.Sources.5.5k" from November 22, 1980. Methods in the file have no class name associated with them. Instead, every method inside a running ST-76 has an offset pointing the beginning of its text in the sources file. Unlike a Smalltalk-80 sources file, a method from the Smalltalk-76 file is not ready to be 'filed in' to a running Smalltalk. The methods in Smalltalk-76 sources are grouped by class, but we did not always know which class it was. It was also hard to tell where one class ended and another began.

The first thing we did was to build a table of all methods in the 5.5k sources file in a modern Squeak. The table had an entry containing the selector, the method source text, and a space for the class. We could write out any table entry as an expression that could install that method in Smalltalk-78. Freudenberg arranged that dropping a file on a web browser running the interpreter placed the file in a list that Smalltalk-78 could see. From a menu inside the running Smalltalk-78, we could "file in" that file.

The selective importer let us bring in source code without changing the bytecodes, but we still worried about two methods in different classes having wildly differing comments. Methods named comment were a prime example, but also +, −, /, =, copy, to:, and printOn:. There were 89 of these ambiguous selectors in Smalltalk-78. We wrote out every (class, selector) pair and included a crude hash for its bytecodes. This allowed us to detect same-name methods where both versions would pass the decompile test.

For the first round, we wrote out all of the methods in Smalltalk-76 that would be non-ambiguous in Smalltalk-78. We still did not know which of several implementations of a selector in Smalltalk-76 was the right one. We simply wrote out all of the implementations! For a given selector, we sent every source version to every class that had it. We depended on the importer to accept only the correct version for each class. This worked. We collected the accepted methods in changeSets and wrote them to files. We sent 1644k of source, of which 241k was accepted.

Back in Squeak, we parsed the Smalltalk-78 fileouts to discover what class had claimed each version of a method. We wrote those class names into the table of Smalltalk-76 methods.

How could we get the right class attached to source code for the ambiguous selectors? The code for each class was contiguous in the Smalltalk-76 sources file. We assumed that for every ambiguous selector, at least one other method in that class was unambiguous. If we started at an ambiguous method, and looked in both directions in the Smalltalk-76 table, we would come across a class name in each direction. If the two classes were different, we simply sent two copies of the method to the merge test. We sent 20k of sources to Smalltalk-78 for the ambiguous selectors, and 16k was accepted.

The Moldable Inspector: a framework for domain-specific object inspection

Andrei Chiş
University of Bern, Switzerland
andrei@iam.unibe.ch

Tudor Gîrba
CompuGroup Medical Schweiz AG
tudor@tudorgirba.com

Oscar Nierstrasz
University of Bern, Switzerland
scg.unibe.ch/oscar

Abstract

Answering run-time questions in object-oriented systems involves reasoning about and exploring connections between multiple objects. Developer questions exercise various aspects of an object and require multiple kinds of interactions depending on the relationships between objects, the application domain and the differing developer needs. Nevertheless, traditional object inspectors, the essential tools often used to reason about objects, favor a generic view that focuses on the low-level details of the state of individual objects. This leads to an inefficient effort, increasing the time spent in the inspector. To improve the inspection process, we propose the *Moldable Inspector*, a novel approach for an extensible object inspector. The Moldable Inspector allows developers to look at objects using multiple interchangeable presentations and supports a workflow in which multiple levels of connecting objects can be seen together. Both these aspects can be tailored to the domain of the objects and the question at hand. We further exemplify how the proposed solution improves the inspection process, introduce a prototype implementation and discuss new directions for extending the Moldable Inspector.

Categories and Subject Descriptors D.2.6 [*Software Engineering*]: Programming Environments—integrated environments, interactive environments

General Terms Tools, Languages, Design

Keywords Object inspector, Domain-specific tools, User interfaces, Programming environments

1. Introduction

Objects integrate data and behavior to model relevant concepts from application domains. Computation is further expressed in terms of interactions between objects. Therefore, understanding objects along with their relationships is critical for developing and evolving object-oriented applications.

Due to their nature, object-oriented applications have a dual representation: static, in terms of source code, and dynamic, in terms of objects. Developers often focus on the static source code to gain insight into the dynamic representation, however, due to mechanisms like inheritance, polymorphism and dynamic binding, understanding objects and relations between objects based only on a static view of the code poses many difficulties [5].

Object inspectors offer a better alternative as they enable developers to explore the actual run-time objects. Inspectors offer generic mechanisms to display and explore the state of an arbitrary object, however they do not take into account the varying needs of developers that could benefit from tailored ways to view and explore object state.

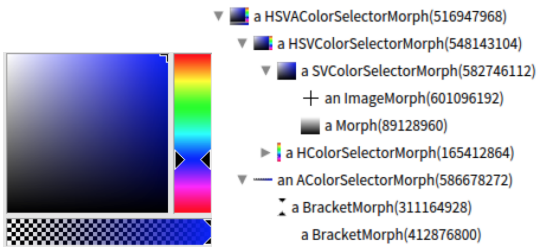
Consider the question of determining whether or not a graphical component that has a certain visual characteristic is present within a list of graphical components. A visual representation of the graphical component provides more insight than just looking at the state of that component. A different question from a different domain consists in determining if an object representing a directory contains a particular file. An object inspector showing the list of files contained by the directory object can provide a straightforward answer.

A generic solution focusing only on object state, while universally applicable, fails to highlight what aspects of an object are important in a given development context. By a development context we understand a specific run-time question from a specific domain (*e.g.*, fixing a performance problem in a parser, finding a memory leak in a graphical framework). This mismatch increases the inspection time as developers have to manually search for what is relevant for their particular contexts.

Traditional object inspectors exhibit this problem since:

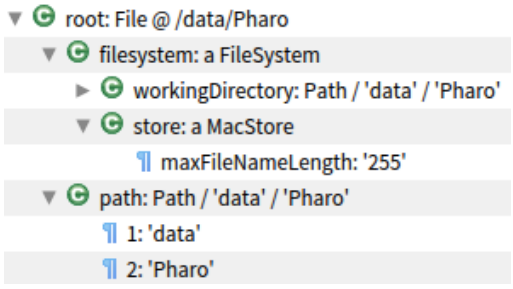
- They rely on predefined, generic state-based presentations for displaying objects, thus ignoring significant differences between objects in different domains;
- They focus on individual objects, thus providing only rigid mechanisms for exploring relations between objects.

These problems can be solved if instead of using a generic object inspector a developer relies on an object inspector that can easily be adapted to the development context at hand (*i.e.*, both the application domain and the developer question). We consequently propose the Moldable Inspector, a



(a) (b)

Figure 1: Two distinct way to look at a morph object for choosing colors depending on the developer needs: (a) presentation showing the visual appearance; (b) presentation showing the structure (the tree of submorphs).



(a)

Name	Size	Creation
..	136	2013-04-02 14:10:55
pharo-vm	272	2014-06-18 15:27:47
.DS_Store	6148	2014-06-18 15:27:36
pharo	382	2014-06-18 15:27:49
pharo-ui	371	2014-06-18 15:27:49
Pharo.changes	17689307	2014-06-18 02:21:46
Pharo.image	43604012	2014-06-18 02:20:36
PharoDebug.log	20717	2014-06-18 15:30:02
vizualization.png	5876	2014-06-18 15:29:14
Workspace.st	73	2014-06-18 15:32:25

(b)

Figure 2: Two distinct ways to look at a directory object depending on the developer needs: (a) presentation showing the state of the object; (b) presentations showing the contained files and directories

novel approach for an extensible object inspector that (1) allows developers to inspect at objects using multiple interchangeable presentations, and (2) provides a workflow in which multiple levels of connecting objects can be seen together, and navigation between objects is guided by the domain and the question. The Moldable Inspector further relies on the idea of using code to both steer the inspection process and to extend the existing presentations at inspection time.

To validate the proposed approach and show that it has practical applicability, we are developing a prototype implementation in Pharo[1], a modern Smalltalk environment. The current version of our implementation features most of the core functionality of the proposed framework in less than 500 lines of code. Furthermore, it has been used to create more than 70 extensions requiring, on average, 8 lines of code per extension. Its small size has two practical advantages: on the one hand it makes it easy to understand; on the other hand it makes the adaptation of the inspector to new run-time questions and domain objects affordable.

The contributions of this paper are as follows:

- Introducing the Moldable Inspector framework for defining a context-aware object inspector;
- Presenting the current prototype instantiation of the Moldable Inspector and discussing several implementation aspects;
- Proposing new directions for extending the Moldable Inspector framework.

2. Why basic inspectors are not enough

To successfully answer a run-time question in an object-oriented system, developers have to identify which objects are relevant for that question and understand those objects along with their interactions. One can anticipate neither what objects will be needed, nor what aspect of an object (*e.g.*, state, code, memory usage, dependencies) will be important for a given question. Thus, object inspectors viewing objects through generic state-based presentations and offering fixed mechanisms for navigating between objects are less suitable for answering run-time questions in object-oriented systems. This section exemplifies these problems.

2.1 Limitations of viewing objects through single views

In Pharo the Morph class is the root class for all graphical components. A morph can contain other morphs (*i.e.*, submorphs). If a developer wants to locate a morph object within a collection, she would benefit from its visual appearance (Figure 1a), but if she wants to debug issues related to the structure of the morph, she needs to see and explore the tree of submorphs (Figure 1b). Both of these representations are valid, but they serve different interests. This type of problem is not isolated and can be found in various other situations.

A different use case consists in inspecting objects modeling various resources. Consider instances of the class FileReference that can refer to a concrete file or directory. During inspection these objects require different representations depending on the developer needs. For example, a state view is sufficient for determining the path of a directory/file (Figure 2a). However, if a developer needs to explore the content of a file/directory, looking just at the state of the corresponding object is not appropriate; a presentation showing the content serves the purpose better. A presentation for a directory (Figure 2b) can show the list of files from

[1] http://pharo.org

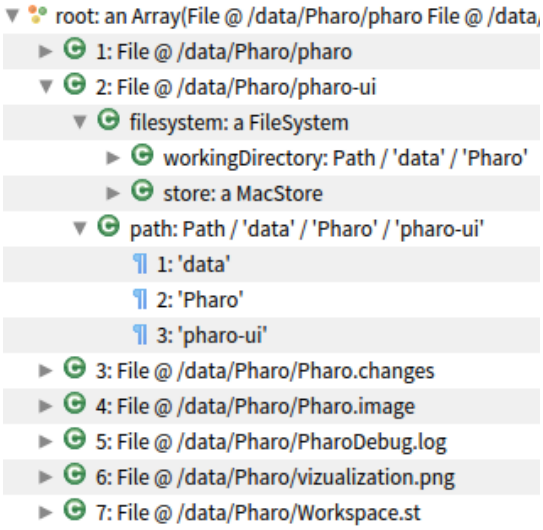


Figure 3: Inspecting a list of files/directories with an object inspector relying on a tree for exploring object state. It is not possible to immediately access the content of a file/directory.

that directory, while one for a file can display its content; the presentation could differ from a file to another depending on the type of the file (*e.g.*, image, text, xml, html). Even if objects representing files and directories are instances of the same class, exploring their content requires different presentations. An object inspector that focuses solely on the state of an object does not support this use case.

2.2 Limitations when focusing on individual objects

On the one hand, visually searching for certain objects within lists (*e.g.*, spotting a particular file/directory inside a directory) requires presentations that make it easy to identify the desired objects (*e.g.*, show the content of a file/directory). On the other hand, this task also requires a developer to simultaneously interact with two objects: the list and an element from that list. Object inspectors that focus on the state of single objects lead to a time consuming exploration effort. Consider using an object inspector representing an object as a tree to look for a particular file/directory within a given directory (Figure 3). This inspector does not provide easy access to the content of a file/directory and further requires a developer to permanently expand and collapse the elements of the target list, increasing the time spent in the inspector.

Another limitation of an object inspector providing only a fixed navigation based on object state is that it does not allow a developer to reach objects not stored in an instance variable of an object already accessible from the inspector. Consider a developer wishing to navigate from a morph representing a list to its context menu (*i.e.*, to check if the context menu has the correct structure). Unfortunately, the context menu is generated on demand every time a user right-clicks on

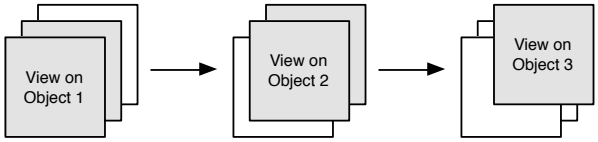


Figure 4: An inspection session consisting of three objects, where each object defines three basic presentations. Gray presentations are valid in one development context, while white presentations are valid in another development context. A *moldable presentation* selects only those presentations that are relevant for the current development context.

an element of the list and is never stored in an instance variable of the morph, thus, it won't be accessible if we can only navigate between objects based on their state. This is an example of a more general problem where references to objects that we wish to navigate to are not actually stored in instance variables of the objects we are currently inspecting.

3. The Moldable Inspector framework

The Moldable Inspector supports developers in reasoning about run-time questions in specific application domains by providing *moldable presentations* and *moldable navigation*. Moldable presentations make it possible for an object to have multiple interchangeable presentations tailored to the domain and the question at hand. Moldable navigation provides a workflow in which multiple levels of connecting objects can be seen together and navigation between objects is guided by the domain and the question at hand.

3.1 Moldable presentations

Reiss argues that software understanding requires custom visualizations tailored to the problems at hand [8]. In the context of object inspectors we argue that understanding objects requires presentations tailored to both the domain and the question at hand (*i.e.*, the development context). While different objects require different presentations, given that they model different entities, the same object requires multiple presentations that depend on multiple usage contexts.

To address this, the Moldable Inspector allows an object to define a set of multiple interchangeable presentations capturing interesting aspects of that object in various development contexts. We will refer to these presentations as *basic presentations*. A developer can then inspect an object using a *moldable presentation* that selects only those basic presentations that are suitable for the current development context.

Moldable presentations are made possible by the Moldable Inspector reifying the current development context (*i.e.*, the domain and the question). An object can thus define a wide set of presentations, not all relevant to a particular context, however, in a given development context a moldable presentation only shows those presentations relevant for that context (Figure 4).

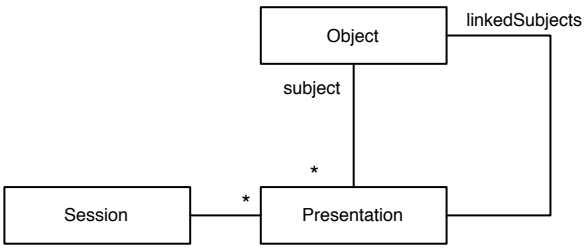


Figure 5: The model behind the Moldable Inspector framework: the *Session* objects reifies the development context; objects define multiple presentations; presentations can indicate relevant objects; when used, only those presentations relevant to the current development context are selected.

3.2 Moldable navigation

Understanding a run-time question involves reasoning about multiple objects and exploring connections between objects. Different types of questions require different kinds of interactions depending on the relationships between objects, the domain and the differing developer needs. Viewing one object at a time and hardcoding the reachable objects by only supporting state-based navigation does not support well this activity.

To overcome these limitations the Moldable Inspector offers *moldable navigation*, *i.e.*, a workflow in which multiple levels of connecting objects can be seen together and navigation between objects is guided by the question and the domain. This navigation mechanism is obtained by allowing each basic presentation to indicate a set of relevant objects that could be inspected next (Figure 5). Since objects are displayed during an inspection session using moldable presentations, only basic presentations relevant to the current question and domain are accessible; thus, the objects indicated by those basic presentations will also be relevant in that situation. However, as one cannot anticipate all developer needs, code can be used to steer the inspection process on-the-fly and increase the set of reachable objects.

4. Implementation aspects

In this section we present the current prototype instantiation of the Moldable Inspector, called the *GTInspector*, and discuss several implementation aspects. The GTInspector is integrated into Moose[2], a platform for data and software analysis [6].

4.1 Supporting moldable presentations

A moldable presentation consists of a set of basic presentations selected according to the current development context. This feature requires mechanisms to associate basic presentations with objects, specify a development context, and filter basic presentations based on the development context.

[2] `http://moosetechnology.org`

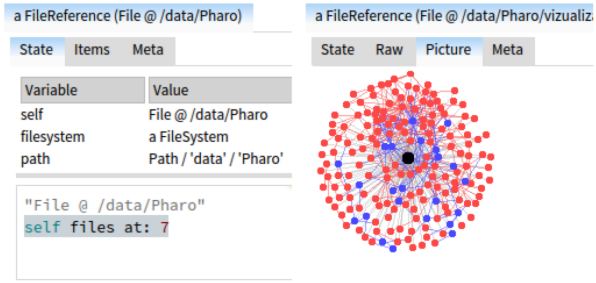


Figure 6: GTInspector displaying a directory and one contained file. Both objects have presentations to display the state and the source code of their class. The directory further has a presentation for showing the contained files/directories ("Items"). The file represents an image and has a presentation that shows its content as text ("Raw") and one that displays the actual image contained in that file ("Picture").

Once a moldable presentation has been computed a solution that can display multiple basic presentations is required.

In the current implementation a basic presentation is associated with an object by defining in the class of the object a method constructing the presentation and marking it with a predefined annotation. Currently defining a development context and filtering presentations based on it is not supported. Instead, regardless of the context, an object is represented using all the available presentations. An object with multiple basic presentations is displayed using a tabulator widget, where each presentation is added as a tab. This allows for interchangeable presentations and also gives an overview of all the basic presentations available for an object. Figure 6 shows how a directory and a file object are represented using this approach.

By default two basic presentations are added to every object: one shows the state of the object while the other gives access to the source code of the object class. By making the state of every object available, we support the classic way of using an inspector. Furthermore, by making the class of the object immediately browsable the inspector supports a common use case of looking implementation up during inspection time.

4.2 Supporting moldable navigation

Providing support for navigating between objects requires a mechanism to show connections between objects. GTInspector shows connections between objects using the Miller columns technique[3]: the next object is always shown to the right. This preserves the entire logical flow of how the developer got to an object. At any moment a predefined number of columns (*i.e.*, objects) is visible and a scroll bar is used to access previous columns.

To support *moldable navigation* we need to allow basic presentations to indicate relevant objects and support developers in using code to guide the inspection process. To sup-

[3] `http://en.wikipedia.org/wiki/Miller_columns`

port the first aspect GTInspector allows developers to continue the navigation by selecting any object available in the current presentation. The second aspect is also supported as the presentation showing the state, available for every object, can be used to write code executed in the context of an object. This can be seen in Figure 6 where the object on the right was obtained by executing the code "self files at: 7" on the object on the left.

4.3 The cost of new presentations

GTInspector has been used to create over 70 basic presentations for 40 different types of objects. On average a basic presentation requires 8 lines of code. Their small size makes them easy to understand and makes the creation of new basic presentations an affordable activity. To give a feeling of the amount of code required to create a new presentation the following lines show how to specify a tree presentation displaying the structure of a morph:

```
1 composite tree
2   title: 'Submorphs';
3   rootsExpanded;
4   display: [:rootMorph | {rootMorph}];
5   format: [:morph | morph printString];
6   children: [:morph | morph submorphs];
7   when: [:morph | morph submorphs notEmpty]
```

5. Improving the inspection process

To show that the GTInspector improves the inspection process in this section we look at how it addresses the limitations of traditional object inspectors encountered in Section 2.

5.1 Multiple presentations for objects

Section 2 showed that depending on the development context one needs to see either the visual appearance of a morph or its structure. The GTInspector addresses this requirement as it can provide two basic presentations capturing these two aspects (Figure 7). It can further provide dedicated presentations for inspecting FileReference objects based on the type and content of the object. For example, a FileReference object representing a directory has a presentation showing list of files/directories within that directory (Figure 8). FileReference objects representing files have dedicated presentations that display the content of the file in a proper way (*e.g.*, a file storing a picture is displayed using a visualization – Figure 6, while a file representing a script using an editor with proper syntax highlighting – Figure 8.)

5.2 Flexible navigation

Visually searching for particular objects within lists becomes possible with the GTInspector: while iterating over the list elements one can obtain a moldable presentation showing each element. For example, to locate a file based on its content one can iterate over the files of a directory and view each file using a specialized presentation (Figure 8).

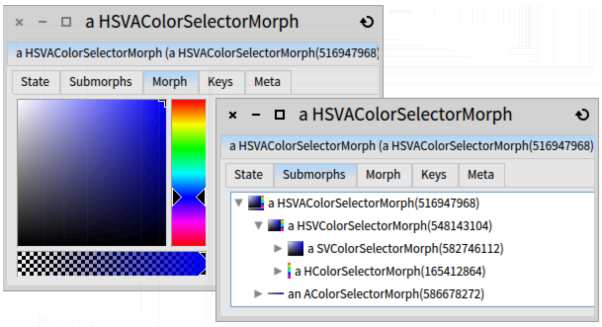


Figure 7: Two different ways to look at a morph for choosing colors.

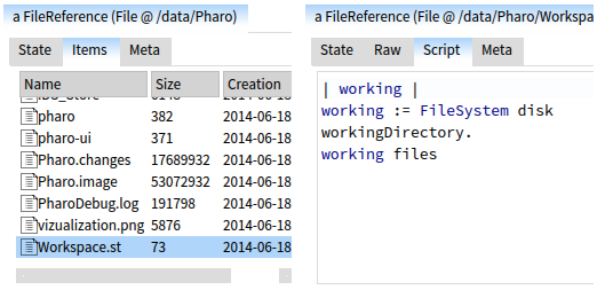


Figure 8: Exploring the content of a directory.

Using code to guide the navigation process makes it possible to reach objects not directly stored within instance variables: Figure 9 shows how one can obtain the context menu of a list morph showing file objects, execute an action from that menu and inspect the result. First the context menu is extracted using the code "self getMenu: false" (the false value indicates the shift key was not pressed). As the menu is a morph we can inspect it visually. Then we can execute the last action, "Copy", which copies a textual representation of the selected object to the clipboard, and finally inspect the current value from the clipboard to see if it is correct.

6. Further directions

While the GTInspector supports a fully functional object inspector there are a number of directions that can be explored in order to further improve it. These include, but are not limited to: identifying common types of recurring run-time scenarios and determining types of basic presentations useful in for addressing those scenarios, modeling the history of an inspection session as a first class entity, improving navigation through large inspection sessions.

6.1 Identifying recurring run-time scenarios

Currently the GTInspector allows objects to have different representations in different development contexts. However, the responsibility of deciding which presentation is relevant in the current development context falls solely on the devel-

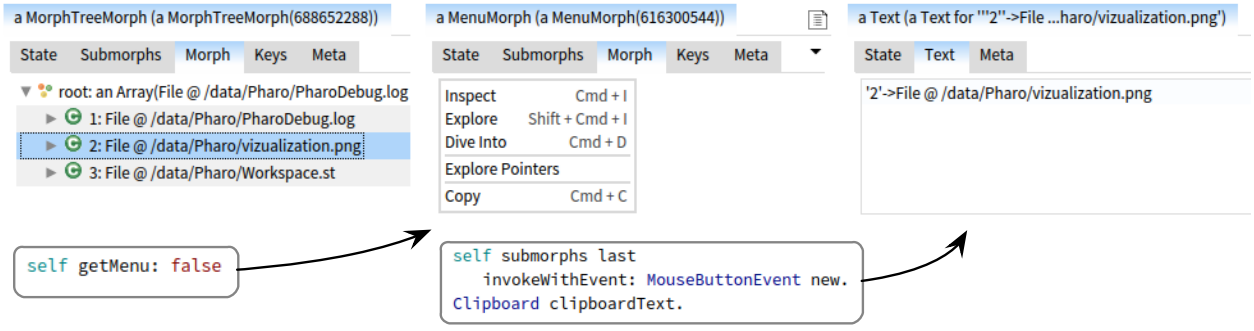


Figure 9: Performing a navigation scenario involving objects that are not directly linked through instance variables: extract the context menu of a list morph, run an action from the menu and verify the result. The first pane shows the tree morph, the second the context menu and the last the current value from the clipboard. At each step code is used to navigate to the next object.

oper. Identifying a set of common scenarios and determining which presentations are useful in those scenarios would lift part of this burden from developers: with the common use cases already supported, developers would only need to focus on creating presentations for their specific situations.

6.2 The inspection session as a first-class entity

While the GTInspector provides support for moldable navigation one has to manually repeat inspection sessions. Modeling the history of an inspection session as a first-class entity would make it possible to store, find and reuse inspection sessions.

6.3 Navigation improvements

While Miller columns are intuitive to use, they have two main drawbacks: *(i)* they require horizontal scroll bars to show deep hierarchical structures (*e.g.*, a deep inspection session) and *(ii)* they do not indicate the relation between two columns (*e.g.*, what did the user do to navigate from one object to the other). We are currently investigating how to solve these problems by providing a new type of scrolling widget showing an overview of the entire inspection session and indicating the relation between columns.

7. Related work

There is a wide body of research looking at how to improve the effectiveness of comprehension and development tools by finding and highlighting contextual information and providing better support for exploring code and data.

Code Bubbles brings the idea of a session of inspection to code understanding and debugging [1, 4]. The approach shows the related entities next to one another and allows the developer to manipulate and store them in sessions. However, this approach still relies on single representations for each entity regardless of the context, and object inspection is particularly only offered through a classic tree like view.

While the focus of our paper is on the conceptual structure of an inspector, the actual implementation is still an interesting aspect that deserves a discussion. The rendering offered by Code Bubbles allows the developer to manipulate a tree, rather than only a list. On the one hand, this is a powerful tool to understand more complicated scenarios. On the other hand, it is a more complicated interface that relies on the developer to organize the bubbles. Our implementation relies on a Miller columns design that requires little space and little spatial maintenance effort from the developer.

jGRASP is an integrated development environment providing object viewers that like our approach allow objects to have multiple presentations [3]. However, *jGRASP* always shows the same objects through the same views as it does not take into account the development context in which those objects are encountered.

Eclipse[4] allows developers to create custom textual representation for objects using "Detail Formatters". Each class can have a Detail Formatter consisting of a snippet of code that constructs a custom string value used to display instances of that class. NetBeans[5] and IntelliJ[6] allow developers to attach multiple such formatters to a given class and switch between them at run time. Unlike the Moldable Inspector these approaches only allow objects to have text representations.

8. Conclusions

Different types of questions exercise different aspects of an object and require different kinds of interaction depending on the relationships between objects, the application domain and the differing developer needs. To support this we presented a novel approach, called the Moldable Inspector, for developing an extensible object inspector that can be adapted to both the objects of a domain and the questions at hand. The development context is reified and used to both select presentations and steer the navigation between objects using a workflow in which multiple levels of connecting objects

[4] `eclipse.org/ide`

[5] `netbeans.org`

[6] `jetbrains.com/idea`

can be seen together. To show that the Moldable Inspector has practical applicability we presented the GTInspector prototype, and discussed several scenarios in which it improves the inspection process.

The Moldable Inspector is part of a broader work on meta-tooling (*i.e.*, tools for building tools) that aims to enable developers to quickly and effectively customize the IDE to suite their development contexts [7]. The Moldable Inspector follows on the Moldable Debugger [2] work that proposed a new approach for developing domain-specific debuggers.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Agile Software Assessment" (SNSF project No. 200020-144126/1, Jan 1, 2013 - Dec. 30, 2015). We also thank Erwann Wernli and Jorge Ressia for their comments.

References

[1] Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr. Code bubbles: a working set-based interface for code understanding and maintenance. In *CHI '10: Proceedings of the 28th international conference on Human factors in computing systems*, pages 2503–2512, New York, NY, USA, 2010. ACM. doi:10.1145/1753326.1753706.

[2] Andrei Chiş, Oscar Nierstrasz, and Tudor Gîrba. The Moldable Debugger: a framework for developing domain-specific debuggers. In *7th International Conference on Software Language Engineering (SLE)*, 2014. to appear.

[3] James H. Cross, II, T. Dean Hendrix, David A. Umphress, Larry A. Barowski, Jhilmil Jain, and Lacey N. Montgomery. Robust generation of dynamic data structure visualizations with multiple interaction approaches. *Trans. Comput. Educ.*, 9(2):13:1–13:32, June 2009. URL: http://doi.acm.org/10.1145/1538234.1538240, doi:10.1145/1538234.1538240.

[4] Robert DeLine, Andrew Bragdon, Kael Rowan, Jens Jacobsen, and Steven P. Reiss. Debugger canvas: industrial experience with the code bubbles paradigm. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 1064–1073, Piscataway, NJ, USA, 2012. IEEE Press. URL: http://dl.acm.org/citation.cfm?id=2337223.2337362.

[5] Alastair Dunsmore, Marc Roper, and Murray Wood. Object-oriented inspection in the face of delocalisation. In *Proceedings of ICSE '00 (22nd International Conference on Software Engineering)*, pages 467–476. ACM Press, 2000.

[6] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*, pages 1–10, New York, NY, USA, September 2005. ACM Press. Invited paper. doi:10.1145/1095430.1081707.

[7] Oscar Nierstrasz and Mircea Lungu. Agile software assessment. In *Proceedings of International Conference on Program Comprehension (ICPC 2012)*, pages 3–10, 2012. doi:10.1109/ICPC.2012.6240507.

[8] Steven P. Reiss. The paradox of software visualization. *VISSOFT 2005. 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, page 19, 2005. doi:10.1109/VISSOF.2005.1684306.

Top-Down Parsing with Parsing Contexts

A Simple Approach to Context-Sensitive Parsing

Jan Kurš Mircea Lungu Oscar Nierstrasz

Software Composition Group,
University of Bern, Switzerland
scg.unibe.ch

Abstract

The domain of context-free languages has been extensively explored and there exist numerous techniques for parsing (all or a subset of) context-free languages. Unfortunately, some programming languages are not context-free. Using standard context-free parsing techniques to parse a context-sensitive programming language poses a considerable challenge. Implementors of programming language parsers have adopted various techniques, such as hand-written parsers, special lexers, or post-processing of an ambiguous parser output to deal with that challenge.

In this paper we suggest a simple extension of a top-down parser with contextual information. Contrary to the traditional approach that uses only the input stream as an input to a parsing function, we use a parsing context that provides access to a stream and possibly to other context-sensitive information. At a same time we keep the context-free formalism so a grammar definition stays simple without mind-blowing context-sensitive rules. We show that our approach can be used for various purposes such as indent-sensitive parsing, a high-precision island parsing or XML (with arbitrary element names) parsing. We demonstrate our solution with PetitParser, a parsing-expression grammar based, top-down, parser combinator framework written in Smalltalk.

Keywords Parsing Expression Grammars, Semi-Parsing, Top-Down Parsing, PetitParser, Context-Sensitive Parsing

1. Introduction

Context-free grammars (CFGs) [1], which are used to describe context-free languages, are very popular among parser developers. There are numerous parsers for a subset of CFGs (LALR, LR, LL and others [2, 3]) and there are techniques for a full set of CFGs as well (GLR [4], GLL [5]).

Parsing Expression Grammars (PEGs) are another formalism for describing languages [6]. PEGs are closely related to top-down parsing and they are syntactically similar to context-free grammars. PEGs can handle some context-sensitive grammars (CSG), *e.g.*, $a^n b^n c^n$ [6], but they cannot handle all of them.

Unfortunately, some computer languages cannot be expressed with either CFGs or PEGs. For example, `C` is not context-free because of its `typedef` feature. Python [7] has a context-free grammar definition,[1] but it requires a special lexer "on steroids" to generate indent and dedent tokens. Many languages might have an ambiguous context-free grammar because of the famous *dangling else* problem.[2] Even an XML-like language with arbitrary element names (contrary to a finite set of element names) cannot be expressed in CFG. The common approaches to overcome the limitations of context-free parsers is to write a parser manually (*e.g.*, Ruby), or add pre-processing (*e.g.*, Python) or post-processing phase (*e.g.*, XML, dangling else problem). Such approaches are not automated and can be time-consuming and error-prone.

In this paper we suggest a simple extension to top-down parsers that allows for context-sensitive behaviour. We propose to extend the input parameter of a parsing function from an input string to a *parsing context*. A parsing context contains an input string, but it can also contain other information. Any parsing function can access whole context information and is allowed to change it. Because the result of parsing can depend on something other than an input stream, we can increase the computational power of a parser. For example, if a parsing context contains a stack we can reach the computational power of a Turing machine [8]. [3]

[1] `https://docs.python.org/3/reference/grammar.html`

[2] `http://en.wikipedia.org/wiki/Dangling_else`

[3] If a pushdown automaton (context-free parser) is extended with a second stack it can simulate a Turing machine. The first stack simulates a tape to the left of the current position, the second stack simulates a tape to the right of the current position.

Yet we don't want to give up the simplicity and comprehensibility of context-free grammars. We therefore keep the context-free formalism (*i.e.*, rules of the form `N ← X`, where N is nonterminal and X is a sequence of nonterminals and terminals), and we hide the context-sensitivity behind nonterminals that refer to parsing functions utilizing parsing contexts. The rules are universal and can be used (and re-used) in multiple grammars.

We have implemented our idea in PetitParser [9], a top-down PEG-based parser combinator [10] framework using packrat parsing [11] written in Smalltalk.

The contributions of this paper are (i) a description of a simple extension that enables the implementation of context-sensitive features in top-down parsers; (ii) an implementation in PetitParser; and (iii) a brief description how to use parsing contexts to implement universal nonterminals for indentation sensitive parsing, XML parsing or high-precision island parsing.

The paper is organized as follows: section 2 describes our extension. Section 3 shows how to implement our extension in PetitParser and discusses the backtracking, memoization and modularity issues. Section 4 presents how to use our extension to implement context sensitive features, namely indentation-sensitive parsing and high-precision island parsing. Section 6 presents the related work and the section 7 concludes this paper.

The implementation of our extension is available online.[4] The reader is invited to download and explore the examples.

2. Parsing Contexts in a Nutshell

The basic idea of parsing contexts is very simple: Use a parsing context as an input to a parsing function instead of an input stream. A parsing context encapsulates an input stream as well as possibly other information. Any parsing function can access the context and can modify it.

This greatly increases the computational power of a traditional context-free parser without introducing hard-to-read grammatical rules. We seek to parse programming languages that are a subset of context-sensitive languages while aiming for simplicity and comprehensibility. For this reason the rules adhere to a context-free formalism. The rules are still in a form `N ← X`, where N is nonterminal and X is a sequence (possibly empty) of nonterminals and terminals. The context-sensitive behaviour is hidden behind universal nonterminals that can be used in various use cases.

Our solution is applicable to any top-down parser. Top-down parsers use backtracking [12], which provides unlimited lookahead, while using memoization (*i.e.*, caching) to avoid exponential complexity that arises when the same text is repeatedly parsed in backtracking alternatives [11, 13]. It is therefore essential for parsing contexts to support these techniques.

[4] `http://smalltalkhub.com/#!/~JanKurs/PetitParser/`

To enable backtracking, it must be possible to remember and restore contexts. When a top-down parser approaches a decision point, the parsing context is saved; then an alternative is selected. If the given alternative proves to be a good one, parsing continues as usually. If the given alternative fails, the parser restores the context (which might have changed while parsing the alternative) and tries another alternative.

To support memoization, a context has to provide a key to the lookup table of cached results. A standard memoizing parser stores the results of parsing under a key consisting of an $(\mathit{input}, \mathit{position})$ pair. With parsing contexts, the key becomes $(\mathit{input}, \mathit{position}, \mathit{context})$. This functionality overlaps with the remember and restore functionality used in backtracking.

Parsing contexts do not change the semantics of context-free parsing function (*e.g.*, choice, sequence) and imposes almost no performance overhead.

2.1 Using Parsing Contexts

To access a parsing context we use an anonymous parsing function. Its only parameter is a parsing context. The body of the parsing function can be almost any code in a target language. The contract of the parsing function is to return a consumed input (possibly empty) in the case of a successful parse, or to return failure f in the case of an unsuccessful parse. To define a parsing function p, we use the following syntax `N ←[:context | ... ]`. We extend rules of the form `N ← X` with the variant `N ← p` to define context-sensitive nonterminals.

The idea of parsing contexts emerges in combination with a parsing framework that predefines some important context-sensitive parsing functions. These functions are then referred to by universally applicable non-terminals. Thus, a grammar implementor does not need the specialized form `N ← p` and can stick with the familiar context-free formalism.

Take for example an XML-like language. The rule:

```
R ← '<'ID'>' '</'ID'>'
```

is not context-free if we want arbitrary `ID`s to match (and if there is an unbounded number of possible `ID`s [8]), so it cannot be expressed in a context-free form. A developer has to define a context-free grammar that accepts a superset of XML with any `ID` pairs and implement an extra pass to verify if the `ID` pairs match.

Yet, if a framework predefines context-sensitive nonterminals `OPENTAG` and `CLOSETAG` representing the opening and closing of an XML element, we can simply use these nonterminals. The context-sensitive XML-like grammar looks just like a context-free one:

```
start   ← element
element ← OPENTAG content CLOSETAG
content ← element*
ID      ← letter+
```

```
OPENTAG ← [:context |
            result ← ID parse: context.
            context elemStack push: result.
            ↑ result
          ]

CLOSETAG ← [:context |
            result ← ID parse: context.
            (context popStack == result)
              ifTrue: [
                ↑ result.
              ].
            ↑ Failure
          ]
```

Listing 1. Implementation of `OPENTAG` and `CLOSETAG` using parsing contexts and Smalltalk as an implementation language.

The developer does not need to know that there is a hidden (possibly complex) code using parsing contexts, as Listing 1 shows.

Another example, in the case of C-like languages, is a `TYPEDEF` nonterminal that stores the type name into the *type table* in a parsing context, and a `TYPE` nonterminal that succeeds only if it sees the identifier that is in the *type table*.

In case of Python-like layout sensitive-languages, we can define `INDENT` and `DEDENT` tokens. From the user's point of view, they are just non-terminals and their complexity is hidden.

3. PetitParser Implementation

PetitParser is a popular PEG implementation for Smalltalk (see Appendix A). PetitParser is easy to adapt to parsing contexts. It suffices to change the `parse:` method from:

```
PetitParser>>parse: stream
    ...
```

to the context-aware parsing method:

```
PetitParser>>parse: aContext
    ...

Object subclass: #Context
    instanceVariables: 'stream'

Context>>stream
    ↑ stream
```

For convenience we extend `Context` with the `Stream` protocol as depicted in Listing 2.

Contexts are extensible with the help of a *properties* protocol. Properties are stored in a dictionary instance variable and can be accessed and set via `getProperty:` and `setProperty:to:` methods.

```
Context>>next
    "Mimic stream behaviour"
    ↑ stream next

Context>>peek
    "Mimic stream behaviour"
    ↑ stream peek
```

Listing 2. A `Context` mimicry to provide a `Stream` protocol.

```
Context>>remember
    | memento |
    memento ← ContextMemento new.
    memento stream: stream copy.
    self rememberProperties: memento.

    ↑ memento

Context>>restore: memento
    stream ← memento stream copy.
    self restoreProperties: memento.

Context>>rememberProperties: memento
    properties keysAndValuesDo:
    [:key :value |
        memento setProperty: key
                to: value copy.
    ]

Context>>restoreProperties: memento
    memento propertiesKeysAndValuesDo:
    [:key :value |
        self setProperty: key
             to: value copy.
    ]
```

Listing 3. A memento protocol of `Context`.

To save and restore contexts we apply the memento pattern [14] (Listing 3). To ensure that a memento cannot be accidentally changed, setters and getters are implemented using a `copy`.

We use the memento to support backtracking, and to implement a memoizing parser that adopts the memento as a key to the memoization table (Listing 4).

4. Case Studies

We now present two advanced applications of parsing contexts, one to support indentation-sensitive parsing, and another to support island parsing.

4.1 Indentation Sensitive Parsing

Indentation-sensitivity (used in Python, Haskell, and F#) is an interesting feature that is hard to implement in context-free parser.

```
MemoizingParser>>parseOn: context
    | key result |

    key ← context remember.
    result ← memoTable at: key

    result ifNotNil: [
        ↑ result
    ].

    ...
```

Listing 4. An implementation of `MemoizingParser` compatible with `Context`.

Python uses a special lexer to produce context-sensitive tokens *indent* and *dedent*, that represent an increased or decreased indentation of a first word on a line. Python's lexer maintains a special stack to track indentation levels. As the indents token is recognized a new value is pushed to the stack. As the dedent token is recognized a value is popped from the stack.

PetitParser is designed to build scannerless parsers. As such, it offers no easy way to track indentation levels. By extending PetitParser with parsing contexts, we can easily track this additional information with the help of an *indentation stack*. We define two new parsers, `IndentParser` and `DedentParser` that mimic the Python-like indent and dedent tokens. Indent succeeds if a line starts in a column greater than the current one in the indentation stack. Dedent succeeds if a line starts on the column that matches the one at the top of the stack. See Listing 5 and Listing 6. Both parsers access the `Context` and modify the `#indentation` property containing the indentation stack.

We demonstrate their use in Listing 7 where we define an indentation-sensitive rule `suite`. A `suite` is a block of code whose statements are all at the same indentation level.

4.2 Bounded Seas

Island parsing [15] is a form of semi-parsing used to recognise just certain parts of interest in a source file (*i.e.*, the "islands") and ignore the rest (*i.e.*, the "water"). The traditional approach of island parsing defines water as *"anything if everything else fails"*. Such a water is easy to define but it ignores the structure of a grammar.

To illustrate, consider an XML file with a list of items as in Listing 8. Each item contains a set of values. Suppose the XML file is malformed and contains broken `value` pairs. The island grammar allows the malformed `value` pairs to be ignored, but it cannot say which `item` the `value` belongs to. This problem can be solved by using *bounded seas*.

A *bounded sea* is an expression that searches for an island in a scope limited by a boundaries. Boundaries are ex-

```
PPParser subclass: #IndentParser.

IndentParser>>parse: context
    | column indentation stack |

    " If at the begining of a line"
    " consume leading whitespaces "
    (context isBeginOfLine) ifFalse: [
        ↑ Failure
    ].
    context consumeLeadingWhitespace.

    " Save the current column "
    column ← context stream column.
    stack ← (context propertyAt: #indent).
    indentation ← stack top.

    (column > indentation) ifTrue: [
        stack push: column.
        ↑ #indent
    ].
    ↑ Failure
```

Listing 5. An implementation of `IndentParser` that detects a Pyton-like indent token.

```
PPParser subclass: #DedentParser.

DedentParser>>parse: context
    | column referenceIndentation stack |

    (context isBeginOfLine) ifFalse: [
        ↑ Failure
    ].
    context consumeLeadingWhitespace.
    column ← context stream column.

    " Restore previous column from the "
    " stack and compare with current "
    stack ← (context propertyAt: #indent).
    stack pop.
    referenceIndentation ← stack top.

    (column == referenceIndentation) ifTrue
    : [
        ↑ #dedent
    ].
    ↑ Failure
```

Listing 6. An implementation of `DedentParser` that detect a Python-like dedent token.

```
suite     ← (newline indent
              statement+
             dedent)

indent    ← IndentParser new
dedent    ← DedentParser new
newline   ← #newline asParser
statement ← suite / if / for / ···
```

Listing 7. Grammar for a layout-sensitive `suite` rule.

```
<list>
    <item>
        <value>a</value>
        <value>b <value> <!-- Malformed -->
        <value>c</value>
    </item>

    <item>
        <value>d</value>
        <value>e</value>
    <item>
</list>
```

Listing 8. An example of a XML file to parse.

```
start    ← '<list>'
             item*
           '</list>'
item     ← '<item>' valueSea* '</item'>
valueSea ← ~value~
value    ← '<value>'content'</value>'
content  ← ...
```

Listing 9. A fault-tolerant XML grammar that uses bounded seas.

pressions that appear before and after the island. We use the syntax `~island~` syntax to create a bounded sea from `island`.

To parse a malformed XML file (*e.g.*, the one as in Listing 8) we define a grammar as in Listing 9. `value` from `valueSea` is always searched between `'<item>'` and `'</item'>`. There could be water (*e.g.*, malformed `value`, comments, *etc.*) both before and after `value`. Because the bounded sea never crosses `'<item>'` or `'</item>'` the parser exactly knows which `item` a `value` belongs to.

Such a sea behaviour is not context-free. Boundaries are by definition context-sensitive, because they are basically the rules used before and after a sea. As a result the `valueSea` being called from the rule `start` from Listing 9 fails on input:

'*</item><item><value>a</value>*'

But the very same rule `valueSea` succeeds being called from `R`, where `R ←valueSea`.

As it turns out, a bounded sea can be implemented as a context-sensitive non-terminal using parsing contexts. Parsing contexts are used to keep a stack of invoked rules. Subsequently, a bounded sea can access the stack and use it to compute its boundaries.

5. Discussion

Albeit very simple and straightforward, the current implementation of a parsing contexts is guilty of exposing global state. Presently, parsing contexts behave as global environments, that can be accessed and modified from any rule. It is a matter of our further research to implement parsing contexts that supports reduced visibility of data.

Our solution sacrifices the linear complexity of Packrat parsing [11] to unlimited complexity, depending on the implemented extensions. For example, the complexity of the indentation-sensitive extension is quadratic in the worst case (at each position we can detect at most n indentation levels, where n is a size of an input). The average complexity is probably better, but this is a matter of further research.

6. Related Work

Attribute grammars [16] extend the possibilities of context-free grammars by introducing attributes and by evaluating them in the nodes of an abstract-syntax tree. Parsing contexts resemble attribute grammars with some important differences: (i) parsing contexts do not filter ambiguous results and are therefore suitable even for non-ambiguous grammars such as PEGs; (ii) parsing contexts directly use the attributes to determine a parsing result; and (iii) parsing contexts hide the attributes, so that a grammar looks looks like a normal context-free grammar (without attributes). Parsing contexts still allow for attributes and do not limit their use.

Context-sensitive grammars [1] are primarily used in linguistics, because context-free grammars cannot describe the phenomena of natural language. Yet, the complexity (PSPACE [17]), understandability and poor semantic suitability led developers to alternatives. In order to specify formal properties of a spoken language, Joshi introduced a mildly context-sensitive grammars [18, 19], that are by definition parsable in polynomial time. There are mildly context-sensitive grammars such as tree adjoining grammars [20], linear context-free rewriting systems [21], or multiple context-free grammars [22].

In contrast to other indentation-sensitive approaches, such as Erdweg *et al.* or Adams [23, 24], our solution a) does not extend BNF notation and b) does not require generalized parsing [23]. The solution we present is specific to Python; the general indentation-sensitive extension is described in a bachelor's thesis [25].

7. Conclusion

In this paper we present a simple extension of PetitParser that allows us to add support for a context sensitive behaviour of XML-like grammars, indentation-sensitive grammars and for high-precision and composable island parsing. We extended an input to a parsing function with a parsing context that can contain information other than an input stream. Parsing contexts are suitable for any top-down parsing technique, because they support memoization, backtracking. Parsing contexts in PetitParser are also extensible and backward compatible.

7.1 Future work

In our future work we plan to investigate the capabilities of parsing-context to capture the most common context-sensitive features of programming languages. Furthermore, we plan to investigate a parsing contexts that supports reduced visibility of data.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Agile Software Assessment" (SNSF project No. 200020-144126/1, Jan 1, 2013 - Dec. 30, 2015).

References

[1] N. Chomsky, Three models for the description of language, IRE Transactions on Information Theory 2 (1956) 113–124, http://www.chomsky.info/articles/195609--.pdf.

[2] A. V. Aho, R. Sethi, J. D. Ullman, Compilers: Principles, Techniques and Tools, Addison Wesley, Reading, Mass., 1986.

[3] A. V. Aho, J. D. Ullman, The Theory of Parsing, Translation and Compiling Volume I: Parsing, Prentice-Hall, 1972.

[4] M. Tomita, Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems, Kluwer Academic Publishers, Norwell, MA, USA, 1985.

[5] E. Scott, A. Johnstone, Gll parsing, Electron. Notes Theor. Comput. Sci. 253 (7) (2010) 177–189. doi:10.1016/j.entcs.2010.08.041.

[6] B. Ford, Parsing expression grammars: a recognition-based syntactic foundation, in: POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, New York, NY, USA, 2004, pp. 111–122. doi:10.1145/964001.964011.

[7] Python, http://www.python.org.

[8] M. Sipser, Introduction to the Theory of Computation, 2nd Edition, Course Technology, 2005.

[9] L. Renggli, S. Ducasse, T. Gîrba, O. Nierstrasz, Practical dynamic grammars for dynamic languages, in: 4th Workshop on Dynamic Languages and Applications (DYLA 2010), Malaga, Spain, 2010.

[10] G. Hutton, E. Meijer, Monadic parser combinators, Tech. Rep. NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham (1996).

[11] B. Ford, Packrat parsing: simple, powerful, lazy, linear time, functional pearl, in: ICFP 02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming, Vol. 37/9, ACM, New York, NY, USA, 2002, pp. 36–47. doi:10.1145/583852.581483.

[12] A. Birman, J. D. Ullman, Parsing algorithms with backtrack, IEEE Conference Record of 11th Annual Symposium on Switching and Automata Theory, 1970 (1970) 153–174doi:10.1109/SWAT.1970.18.

[13] P. Norvig, Techniques for automatic memoization with applications to context-free parsing, Computational Linguistics 17 (1) (1991) 91–98, cited By (since 1996).

[14] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley Professional, Reading, Mass., 1995.

[15] L. Moonen, Generating robust parsers using island grammars, in: E. Burd, P. Aiken, R. Koschke (Eds.), Proceedings Eight Working Conference on Reverse Engineering (WCRE 2001), IEEE Computer Society, 2001, pp. 13–22. doi:10.1109/WCRE.2001.957806.

[16] D. Knuth, Semantics of context-free languages, Mathematical systems theory 2 (2) (1968) 127–145. doi:10.1007/BF01692511.

[17] M. Garey, D. Johnson, Computers and Intractability: A Guide to the Theory of NP-completeness, Freeman, San Francisco, 1979.

[18] A. K. Joshi, Tree adjoining grammars: How much context-sensitivity is required to provide reasonable structural descriptions?, Cambridge University Press, 1985.

[19] K. Laura, Parsing Beyond Context-Free Grammars, Springer-Verlag, 2010.

[20] A. Joshi, Y. Schabes, Tree-adjoining grammars (1997).

[21] K. Vijay-Shanker, D. J. Weir, A. K. Joshi, Characterizing structural descriptions produced by various grammatical formalisms, in: Proceedings of the 25th Annual Meeting on Association for Computational Linguistics, ACL '87, Association for Computational Linguistics, Stroudsburg, PA, USA, 1987, pp. 104–111. doi:10.3115/981175.981190.

[22] H. Seki, T. Matsumura, M. Fujii, T. Kasami, On multiple context-free grammars, Theoretical Computer Science 88 (2) (1991) 191 – 229. doi:10.1016/0304-3975(91)90374-B.

[23] S. Erdweg, T. Rendel, C. Kästner, K. Ostermann, Layout-sensitive generalized parsing, in: SLE, 2012, pp. 244–263. doi:10.1007/978-3-642-36089-3_14.

[24] M. D. Adams, Principled parsing for indentation-sensitive languages: Revisiting Landin's offside rule, in: Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '13, ACM, New York, NY, USA, 2013, pp. 511–522. doi:10.1145/2429069.2429129.

[25] A. S. Givi, Layout sensitive parsing in Petit Parser framework, Bachelor's thesis, University of Bern (Oct. 2013).

[26] E. Visser, Scannerless generalized-LR parsing, Tech. Rep. P9707, Programming Research Group, University of Amsterdam (Jul. 1997).

A. PetitParser

PetitParser [9] is a parsing framework using four methodologies: a) Parsing Expression Grammars (PEGs); b) Scannerless Parsers [26] that combine lexical and context-free syntax into one grammar; c) Parser Combinators that are building blocks for parsers modeled as a graph of composable objects (they are modular and maintainable, and can be changed, recomposed, transformed and reflected upon); d) and Packrat Parsers that improve performance of PEGs by using memoization.

A.1 Parsing Expression Grammars

PEGs were first introduced by Ford [6] and the formalism is closely related to top-down parsing. PEGs are syntactically similar to CFGs [1], but they have different semantics. The main semantic difference is that the choice operator in PEG is ordered — it selects the first successful match — while the choice operator in CFG is ambiguous. PEGs are composed using the operators in Table 1.

Operator	Description
''	Literal string
[]	Character class
.	Any character
(e)	Grouping
$e?$	Optional
$e*$	Zero-or-more repetitions of e
$e+$	One-or-more repetitions of e
$\&e$	And-predicate, does not consume input
$!e$	Not-predicate, does not consume input
$e_1\ e_2$	Sequence
e_1 / e_2	Prioritized choice

Table 1. Operators for constructing parsing expressions

A.2 PetitParser in Smalltalk

To create a parsing expression as in Table 1, PetitParser uses internal DSL. In this paper we will use a DSL as in Table 2.

Each of the operators is implemented as a subclass of `PPParser`. `PPParser` contains an abstract method `parse:` that accepts an input as an argument and performs the parsing and returns a result or a failure.

Operator	Description
`'abc' asParser`	Literal string
`#any asParser`	Any character
`#newline asParser`	A new line
`(p)`	Grouping
`p ?`	Optional p
`p *`	Zero-or-more repetitions
`p +`	One-or-more repetitions of p
`p and`	And-predicate
`e not`	Not-predicate
`p1 p2`	Sequence
`p1 / p2`	Prioritized choice

Table 2. Operators for constructing parsing expressions

Towards a new package dependency model

Christophe Demarey
RMoD Project-Team, Inria Lille–Nord Europe
christophe.demarey@inria.fr

Damien Cassou
RMoD Project-Team, Lifl, Université de Lille 1, Inria Lille–Nord Europe
damien.cassou@univ-lille1.fr

Stéphane Ducasse
RMoD Project-Team, Inria Lille–Nord Europe
stephane.ducase@inria.fr

Abstract

Smalltalk originally did not have a package manager. Each Smalltalk implementation defined its own with more or less functionalities. Since 2010, Monticello/Metacello[Hen09] one package manager is available for open-source Smalltalks. It allows one to load source code packages with their dependencies. This package manager does not have all features we can find in well-known package managers like those used for the Linux operating system. This paper tries to identify the missing features and proposes solution to reach a full-featured package manager. A part of this solution is to represent packages and dependencies as first-class objects, leading to the definition of a new dependency model.

Keywords package management system, package manager, dependency

1. Introduction

In this presentation, the *package* term will be used to depict a shippable piece of software (something that you can deliver) and will not be related to system packages nor source code Version Control Systems (*e.g.*, Monticello), two commonly used package meanings in Smalltalk. Smalltalk (Gemstone, Pharo, Squeak) did not have a package manager for years. Indeed, with the image paradigm, the need of a package manager was not seen as a first need. With time, the need of a package manager comes up to have minimal images where you can load only packages you need. In such situations, a package manager is really important to be able to load all required packages (including transitive dependencies). Loading transitive dependencies implies that packages describe their dependencies.

Since 2010, Metacello provides a solution to manage packages and their dependencies [BCDL13]. It was a huge effort towards a modular system where you can load additional application, libraries and their transitive dependencies. Metacello also evolved year after year, adding new features. At this time, we have a better comprehension of what is working with the Metacello package manager and what is missing or should be improved. We can also compare this package manager with other languages. The goal of this paper is to describe improvements, new functionalities we would like to have for the next generation package manager. First, we describe missing functionalities and then propose a solution for each one. To conclude, we depict future work in this area.

2. Problem Description

> "A package management system, also called package manager, is a collection of software tools to automate the process of installing, upgrading, configuring, and removing software packages [...] in a consistent manner. It typically maintains a database of software dependencies and version information to prevent software mismatches and missing prerequisites.", *Wikipedia*

This explanation of a package management system is more operating-system oriented but also applies quite well to a programming language. From a language point of view, what can you expect from a package manager? Here is a non-exhaustive list of wished functionalities for a package manager:

- ensuring system coherence: install, update, remove packages without breaking installed packages. If a conflict occurs, the package manager should explain the conflict and give the user the choice on how to solve it,
- first-class dependencies: packages and their dependencies should be first-class objects, easily accessible from other tools,
- synthetic package descriptions: provide a quick way to create new packages and express their dependencies through short and easily understandable descriptions,
- automated load order computation: automated computation of the load order from the dependency graph.
- knowledge on the system state: a user should be able to query on installed packages in the system,
- knowledge on available packages: a user should be able to query on available packages for the system,
- solving review: allow the user to review what will be installed before actually install packages,
- reproducible loading: once the solving done, the user may want to reuse this solving to load the same set of packages on identical systems (e.g, on the N production servers),
- conditional loading support: filter out packages that cannot be installed in the system (e.g. requires a specific version of the system),
- complex constraints support: declaring a dependency to a fixed version is not enough. A package manager has to allow more constraints like a version range (between version 1.2 and version 1.6, greater than 2.0) or boolean expressions,

- support of update strategies: these strategies will enables automatic updates of compatible packages (e.g., security or bug fixes),
- independence from Version Control Systems: Dependency descriptions should refers to versionned packages and not source code artifacts.

We can see that functionalities expected from a package management system for a programming language are quite the same as functionalities expected for an operating system.

To come back to Metacello, we present an overview of the functionalities offered by package management systems for open-source Smalltalk and some other well-known package management systems.

Functionnality	Metacello	Maven	apt
ensuring system coherence	Partial[1]	Partial[2]	Yes
first-class dependencies	No	No	No
synthetic package descriptions	No	Partial[3]	Yes
automated load order computation	No	Yes	Yes
knowledge on the system state	No	Yes	Yes
knowledge on available packages	No	Partial[4]	Yes
solving review	Yes	Yes	Yes
reproducible loading	Partial [5]	Partial[6]	Yes
conditional loading support	Yes	N/A	No[7]
complex constraints support	No[8]	Yes	Yes
support of update strategies	No[9]	Yes	Yes
independance from VCS	No	Yes	Yes

Table 1. Package management systems functionalities

In the following subsections, we focus on functionalities not yet covered by Metacello, the only tool available for open-source Smalltalks.

2.1 Ensuring system coherence

When installing a new library (or updating an already loaded library) into an image, there is no guarentee that already installed libraries will continue to work. There is no record of which configurations are already installed in the image. Configurations can be collected in the image but it is impossible to know if a particular configuration has been loaded and which version was loaded. The record of such information is very important to allow tools and users to query about installed software. This information is also primordial to install new libraries.

Let's take an example: the package A is already installed in the system and depends on B v1.1. We want to install the package D that has a dependency to B v1.2. There is a conflict: we cannot have both B v1.1 and B v2.2 in the image. At the present time, if we request to install D, B v1.2 will be loaded in the image and A will be broken! This should not happen. Installing a new package into the system should take into account what is currently loaded and what may break! The same problem applies when trying to update an installed package into the system.

[1] Metacello provides hooks to execute code on upgrade or downgrade of an installed package

[2] Maven takes decisions that may lead to an inconsistent system. However, the user can force this decisions.

[3] XML is verbose

[4] Maven has a central repository but does not use it to propose packages.

[5] The use of symbolic versions leads to unreproducible loadings

[6] not possible by using SNAPSHOT dependencies

[7] There is one specific central repository per platform

[8] Only fixed versions supported

[9] Symbolic versions may be used for updates but without warranty on the backward compatibility.

2.2 First-class dependencies

Packages and their dependencies should be first-class objects. The package management system as well as other tools need a quick access to these information. First-class dependencies will facilitate the work needed to go to a modular system, open doors to new tools (e.g. automatic update of dependencies information from the source code). In the current package manager used by open-sources Smalltalk, three kind of dependencies [10] are used to express software dependencies: dependency referring to a project (piece of code with a dedicated description/configuration), to a package and to a group that is a collection of projects, packages or groups. These notions are very close and it is not always easy to understand the subtle differences between these concepts. It will increase the readability to merge them in a common concept. First-class dependencies will also enable the knowledge on the system state (installed packages). With first-class dependencies, we can also add more information on packages, i.e., more meta-data. Currently available meta-data, stored in Configurations, are:

- the package version author,
- the package version description,
- the package version timestamp,
- and the package version blessing: the tag used to manage symbolic versions (development, release).

We would like to add useful information such as the project license, a brief project description, the project website url, the project inception year, a link to the issue management system, a link to the mailing lists, the list of developers/contributors, etc. Other tools (or users) can use such information to choose the package fitting their needs. For example, Maven pom files[11] provide a lot of meta-data for each project.
All this information on packages needs to be loadable easily without loading the package itself and without installing any new code in the system. Indeed, it is very strange to modify the image state by loading new classes (Configurations) to only read some information on packages. It may also be dangerous if the loaded code overrides some existing code in the image. Storing meta-data in methods of a class allows versioning of meta-data only if you use Monticello as Version Control System, and not with any other VCS. VCS are able to version any kind of data: source code, images, text, binaries. Moreover, you can keep each version of your meta-data easily by publishing them to a central package repository, even with Monticello.

2.3 Synthetic package descriptions

Package descriptions are currently cluttered with a lot of specific dependencies hard to handle: platform specific packages and test packages.

2.3.1 Management of platform specific packages

Currently platform-specific code goes into a dedicated package. Then, you still need to tell Metacello which package to load according to the targeted platform. It is done with the for:do: message as shown in the following snippet.

[10] See DeepIntoPharo (http://deepintopharo.com), Managing projects with Metacello chapter

[11] http://maven.apache.org/pom.html

```
spec
  group: 'Core'
  with: #('CoolBrowser-Core' 'CoolBrowser-Platform')
spec
  for: #gemstone
  do: [ spec
        package: 'CoolBrowser-Platform'
        with: 'CoolBrowser-PlatformGemstone' ]
spec
  for: #pharo
  do: [ spec
        package: 'CoolBrowser-Platform'
        with: 'CoolBrowser-PlatformPharo' ]
```

Listing 1. Platform packages management example

This information is redundant and clutters the package dependencies description. It should be simplified. You should just say that CoolBrowser-Core requires a platform specific package. The package manager should be smart enough to choose a package fitting the platform requirements.

2.3.2 Management of test packages

Tests are also often put in a dedicated package to allow the loading of a library with or without tests. Metacello does not provide an option to load (or not) test packages. It implies that the developer has to provide a way to load the code with or without tests by himself in the configuration. Here, again, the package dependencies description will be cluttered by groups defined to load tests or not. To summarize, current package dependencies description is far too verbose and should be simplified.

2.4 Automated load order computation

In current package descriptions, the developer needs to explicitly specify which packages should be loaded before the described package. Indeed, the whole dependency graph may contains cycles. With cycles, it is difficult to know which package to load before the others. Most cycles are introduced by the proliferation of specific packages: tests packages, platform-specific packages. The definition of the packages load order is delegated to the developer. It is more work to maintain a consistent load order and also more error prone. It would be good that the package management system gets this preoccupation to free the developer mind and to lighten the package descriptions.

2.5 Complex constraints and update strategies support

Metacello is a first step towards a better modularity. It enables to load quite easily packages and their dependencies. To achieve that, you need to specify which version of a package you need. Actually, Metacello supports only one kind of constraint: exact version match and allows only one constraint per package. It means you are locked to a specific package version. If you want to use a more recent version, you need to update your configuration. It is not totally true because Metacello allows the use of symbolic versions like #development, #stable or #release. These symbolic versions are useful to easily get the latest stable version of a package or to get the development version. The use of symbolic versions is not restricted to these use cases. For example, you can specify that your package relies on the latest stable version of another package. It introduces some flexibility but not enough. We need to express more constraints on package dependencies like: =1.0, =1.0 or 1.1, <2.0, >2.0 and <2.5, etc. More constraints and more constraint kinds will offer new possibilities but there is a price to pay for that: constraints solving will become more complex and time consuming. With only one possible path to follow (each constraint is solved to a specific version and only this one), the current solving can use an ad-hoc algorithm and be quite efficient. With the introduction of non-fixed dependency version constraints, the number of possible pathes explodes and we need an efficient algorithm. Solving a constraint satisfaction problem on a finite domain is an NP-complete problem in general. It implies to use a dedicated solver for this problem. To introduce automated update strategies, we need to know which versions are only bug fixes versions, which versions are backward compatible and which one are not backward compatible. This information is currently not available for packages.

2.6 Reproducible loading

A package management system has several responsibilities:

- allow the user to express requests (installation, update, removal) on packages,
- find a solution (if any) to the user request,
- and apply this solution.

The solution to a user request, e.g. a package installation, is called a dependency resolution. This dependency resolution should be serializable and reusable. Indeed, the solving can be made one time and the solving result can be used many times in possibly many images. That way, we ensure that the exact same set of packages will be loaded into different images. It is very convenient, for example, to ensure that packages installed in the production image(s) will be exactly the same packages deployed in the development image. Such a solution also allows a decoupling between the solving part and the loading part. You can imagine a minimal image with no solver but able to load already solved dependency resolutions.

2.7 Independence from Version Control Systems

A big drawback with the current description of dependencies, is that descriptions are coupled with the legacy Smalltalk Version Control System: Monticello[12]. Indeed, to express dependencies, you need to reference Monticello zip files (mcz files).

```
spec
    package: 'CoolBrowser-Core'
    with: 'CoolBrowser-Core-BobJones.20'
```

Listing 2. Example of explicit reference to a VCS

In the previous example, CoolBrowser-Core-BobJones.20 refers to the CoolBrowser-Core-BobJones.20.mcz Monticello file. With the emergence of the git[13] Version Control System (VCS), a new way to express dependencies comes up allowing the developer to declare dependencies without specifying a specific version of these dependencies. In fact, the default dependency version is the head of the Version Control System but it can be set by specifying a specific repository URL like bellow:

```
github://demarey/metacello-work:1c8c138a7be...
```

Listing 3. URL used to refer to a specific version in a git repository

Despite the fact that several VCS are supported (git, Monticello, flat files), current package descriptions are closely tied to VCS: in numbered versions, you have to explicitly reference a particular artefact of the supported VCS, in general an mcz file name. We need to find a way to decouple the dependency description, and overall packages distribution from the VCS. We can also ask ourselves Why is it coupled? Pharo does not deliver binary packages (even if it is the

[12] http://www.wiresong.ca/monticello/

[13] http://git-scm.com/

case in professional environments such as VisualWorks [MLW05]) but rather packages with the source code. It can explain why there is a coupling between the package distribution and the Version Control System, but it should not be coupled! Source code versions are not the same concept as deliverable package versions.

3. Proposed Solution

This section will expose solutions for each problem exposed above. Even if each problem is seen as an individual case, all solutions put together describe a coherent approach.

3.1 Ensuring system coherence with installed package information

The solution to avoid broken libraries after installing/updating a software is very simple: we need to keep information about installed packages / software into the image. With a good object model of these dependencies, other tools will be able to use this information to ensure the coherence of the system. The simple proposition is to create a Package Registry with the responsibility to register all packages loaded into the image, and of course all meta-information on these packages. This registry will be used by other tools to ask for installed software, but also to get input for dependency solving.

For example, we can imagine different strategies to solve a software installation. Indeed, a sofware installation request can be translated to a constraint satisfaction problem. One strategy to solve a software installation could be to *minimize the number of updated packages and the number of new packages to install*. To implement this strategy, you need to know what is already installed in your system. It takes more importance when you need to install a software without breaking those already installed. If we use the example exposed below: *package A is already installed in the system and depends on B v1.1. I want to install the package D that has a dependency to B v1.2.*, if A can only use the version 1.1 of the package B, we need to add this constraint before starting to solve dependencies. The package registry will help to find the constraints we need to add to the dependency solving to keep the system in a coherent state, i.e., with all packages / software working.

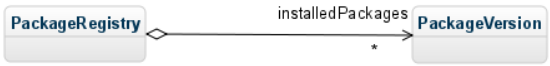


Figure 1. Package registry

We can imagine that such package information can be stored in the package Manifest (a package manifest is data class storing information about rule false positives) and extracted on demand to be published on package catalog and other external package description systems.

3.2 First-class dependencies

Having first-class dependencies in the image implies to extract the core concepts manipulated by software dependencies. Is a dependency to a package of your project the same kind of a dependency to a package outside your project? A package represents a piece of software you want to distribute or use. This piece of software may be something you developed or something coming from outside your project. There is not really a difference. Then, how to represent a group of pieces of software, i.e. a group of packages? A group is just a meta-package: a meta-package does not contain actual software, it is an empty package that simply depends on other packages, thus forming a group. With packages and meta-

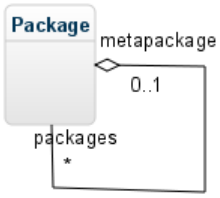


Figure 2. MetaPackage representation

packages, we have an uniform representation of dependencies.

Instances of the model introduced below describe packages with information on dependencies but also other meta-data. Package descriptions will be represented with objects in the image but we also need to store them with the source code. To achieve that, we need to serialize and also, deserialize these objects. Package meta-data needs to be easily accessible without being obliged to load the package itself or to install new source code in the image. To reach this goal, we need to define a serialization format for these meta-data. This format should be easily loadable and saved into/from the image and, if possible, easily human-readable. A good solution is to serialize packages meta-data with STON[14], a Smalltalk variant of the well-known JSON standard. STON is quite readable, close to a standard and provides automatic serialization/deserialization of objects.

Here is an example of what could be a serialization of a package metadata:

```
Package {
  #name : 'Seaside',
  #version : 3.1.0,
  #description : 'The framework for developing
    sophisticated web applications in Smalltalk.',
  #website : 'http://www.seaside.st',
  #dependencies : {
    'Grease' : 1.1,
    'Seaside-Core' : 3.1.0,
    'Seaside-Canvas' : 3.1.0,
    'Seaside-Session' : 3.1.0,
    'Seaside-Component' : 3.1.0,
    'Seaside-RenderLoop' : 3.1.0,
    'Seaside-Tools-Core' : 3.1.0,
    'Seaside-Flow' : 3.1.0,
    'Seaside-Environment' : 3.1.0,
    'Seaside-Widgets' : 3.1.0
  }
}
```

Listing 4. New serialization example of a package metadata

As package meta-data are outside the image, we need to find a way to store it with Smalltalk source code for legacy VCS (e.g., Monticello). The easiest solution would be to include the STON file into the mcz file that is a zip file but this solution will imply to transfer the whole mcz file to only get the metadata. It may be slow with a low bandwidth. Another option could be the creation of a specific Monticello package to hold these meta-data. The specific mcz will contain nothing but the STON file and Monticello meta-data. This way, the solution is still compatible with Monticello and can retrieve packages meta-data quite efficiently. However having empty package from a programmer point of view can be confusing.

[14] https://github.com/svenvc/ston/blob/master/ston-paper.md

Managing two packages for a package and its description is not optimal.

3.3 Synthetic package descriptions

With first-class dependencies, we have a nice dependency model in the image but we still need to find a way to handle platform-specific packages. The Debian operating system introduced the notion of virtual packages[15] for its package management system.

> A virtual package is a generic name that applies to any one of a group of packages, all of which provide similar basic functionality. For example, both the tin and trn programs are news readers, and should therefore satisfy any dependency of a program that required a news reader on a system, in order to work or to be useful. They are therefore both said to provide the virtual package called news-reader.

A virtual package is some kind of under-specified contract. Indeed, the contract only relies on the virtual package name and has no description. Some packages require this contract and some others provide it. Virtual packages should be used carefully because there is no verification that a package really implements the contract needed. There may also be some naming comflicts if appropriate names are not chosen. Beside that, virtual packages offer great features such as a loose coupling between packages. The package manager can choose the best package providing a virtual package according to the specific user request and environment. This low coupling avoids to predict all potential cases in the package description.
The idea is to use virtual packages to manage platform-specific packages. If a package Foo needs platform-specific packages, then it should declare a dependency to the Foo-Platform virtual package. The package implementor then needs to create platform-specific packages (e.g., Foo-Pharo, Foo-Gemstone), each providing the Foo-Platform virtual package. At the solving time, the package manager will search in the repository for all packages implementing the required virtual package. Of course, a virtual package will also have a version to choose an appropriate version of the virtual package. To work properly, packages (and as a consequence virtual packages) need to define requirements. Those requirements will be checked to see if a package can be installed on a given platform (e.g., Foo-Pharo requires the Pharo platform). Requirements already exist with Metacello and are named platformAttributes. By checking package requirements, the package manager will see that the package Foo-Gemstone cannot be installed on Pharo, and then the Foo-Pharo will be selected. The package description becomes shorter and cleaner.

Here is an example of a legacy description:

```
spec for: #common do: [
  spec
    package: 'Foo' with: 'Foo-Platform';
    group: 'default' with: #('Foo' 'Foo-Platform') ].
spec for: #gemstome do: [
  spec
    package: 'Foo-Platform' with: 'Foo-Gemstone'].
spec for: #pharo do: [
  spec
    package: 'Foo-Platform' with: 'Foo-Pharo'].
```

Listing 5. Legacy description for platform-specific packages

The Foo package becomes:

```
spec
  requires: 'Foo-Platform'.
```

Listing 6. Foo package description

The platform-specific package becomes:

```
spec
  provides: 'Foo-Platform'.
```

Listing 7. Foo-Pharo and Foo-Gemstone package description

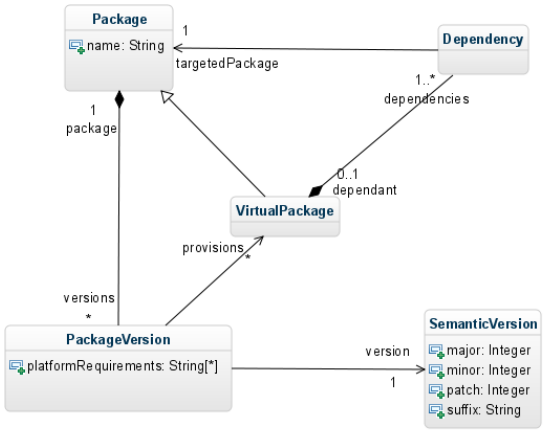


Figure 3. Virtual package modelization

With the unification of dependency description and the introduction of virtual packages, descriptions become smaller and easier to read, write or maintain.

To handle properly test dependencies (but also other kind of dependencies like development dependencies), we propose to define a scope to dependencies. A dependency may be needed to run tests but not at runtime, another could only be useful to develop the package. A dependency scope is in fact a kind of dependency. We propose to define a core dependency class and specialized versions of this dependency: *runtime dependency*, *test dependency* and *development* dependency.

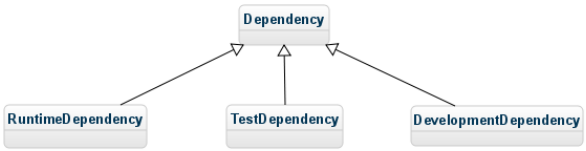


Figure 4. Dependency scopes

3.4 Automated load order

The biggest problem to enable automated load order computing by the package management system is the presence of cycles in the dependency graph. The analysis of several projects such as Seaside [DRS+10] showed that most cycles involve platform-specific packages. If we omit these dependencies, it is hard to find a cycle in a dependency graph. The solution proposed is to consider platform-specific packages as part of the core package.

Let's take an example: there is a Foo package. If we find a Foo-Tests package, we should consider that this package is part

[15] http://www.debian.org/doc/manuals/debian-faq/ch-pkg_basics.en.html

of the Foo package. The platform-specific package is in a different package for technical reasons (selective loading) but conceptually, the platform-specific and the core packages represent the same package. At the loading time, this conceptual representation will be translated into a batch loading of these packages. Indeed, they depend on each other, and then should be loaded at the same time. With this approach cycles should be removed from the dependency graph. If there are still some cycles, it may highlight a design problem.

3.5 Complex constraints solving and update strategies support

To be able to express more sophisticated constraints than *'I depend on the package foo in the exact version 1.1'*, we need to revisit the dependency description format to allow more expressivity. For example, if I need a version of B at least equals to the version 1.1, I will write B >1.1. Such constraints will imply to solve NP-complete problems and thus to use proper solvers. To ease package updates and packages descriptions, we can also take advantage of the semantic versioning[16]. It is nothing else that a convention to follow to number versions of packages. By following these conventions, we will be able to perform automatic updates like bug fixes updates, security issues update because version numbers will give us information on backward compatibility. For example, v1.2.4 will be compatible with v1.2.3, v1.2.1 and v1.2.0 (bug fixes versions) but also with the v1.* versions. On the other side, it will not be compatible with the v2.* versions. More sophisticated constraints and the adoption of a versioning strategy: semantic versioning will open new doors to dependency management. It should lead to less package versions, and at least less package description. It is also important to notice that tools can help to ensure the coherence of the versioning strategy (e.g., forbid the use of minor/patches version if an API change is detected).

3.6 Reproducible loading

To enable reproducible loading, we need to serialize dependency resolution. Such a file is named a lock file in the Composer dependency manager (for the PHP language). Composer writes the list of the exact versions it will install into a composer.lock file. It locks the project (package) to those specific versions. This mechanism is useful to save resolution time and to be sure to install exactly the same set of packages on the same machine or on other machines. It is similar to the Snapshotcello behavior which freezes the versions. It also enables the installation of packages into an image that does not have a solver.

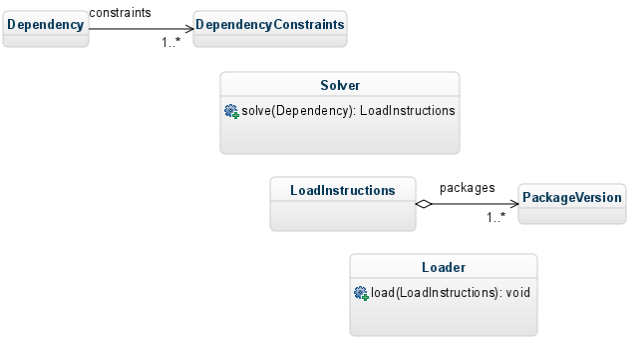


Figure 5. Solver and Loader decoupling

[16] http://semver.org/

As you can see on the class diagram, a Solver will take as input a dependency that is in fact a constraint or a set of constraints on one or more packages. The result of a solving is a LoadInstructions object having an ordered collection of all packages (with their specific versions) to load. This object may be serialized (or not) and then given to the package loader to actually perform the load. We propose to use the STON format to serialize LoadInstructions objects. The STON format will be already used to store package meta-information.

3.7 Independence from Version Control Systems

The best way to decouple package distribution from the source code / VCS is to set up a package repository where packages will be published. Published packages will have an independent version numbering. Many languages adopted this approach Java with the Maven central repository[17], Ruby with RubyGems[18], Perl with CPAN[19], Python with PyPI - the Python Package Index[20], PHP with Packagist[21], JavaScript with Bower[22], ... This approach allows a great decoupling but also open doors for added-value features:

- a central place for the community to share artifacts,
- a central place to search for existing libraries,
- a central place to find (meta-)information on libraries.

For such a service, a preoccupation may be the duplication of the source code (in the VCS and in the central repository) and the storage needed. Both issues can be solved easily: there is no need to duplicate data. We can simply publish a description of the package version in the package repository. This description will contain all information needed to get sources directly from any VCS.

4. Conclusion

In this paper, we looked at the package management system challenges. We proposed different solutions to address these problems:

- have first-class objects for package dependencies in the system,
- use virtual packages to handle the complexity of platform-specific code,
- set up a repository dedicated to host packages and decoupled from Version Control Systems,
- support complex constraints support,
- adopt the semantic versioning.

The adoption of these propositions will increase the solving complexity[23] but, on another side, will offer a lot of facilities and new functionalities to the developer. Package descriptions will be easier to write and maintain, we will be able to do automatic updates, check the system coherence, and last but not least get a central repository for Pharo/Gemstone libraries. It will enhance the

[17] http://search.maven.org

[18] https://rubygems.org/

[19] https://metacpan.org/

[20] https://pypi.python.org/pypi

[21] https://packagist.org/

[22] http://bower.io/search/

[23] when a user request a package installation or update, the package manager needs to find a solution (a set of package versions to install) fitting expressed constraints (direct dependencies constraints) and transitive constraints (constraints expressed on transitive dependencies, i.e. dependencies of dependencies of the package to install). If constraints are not strong, the numbre of pathes to explore by the solver is huge and it is more complex to find a solution in a reasonnable time.

share and reuse of libraries by giving visibility on these projects. This model is adopted by a wide range of other languages.

Further work will target complex dependencies solving. There are two solutions: implement a new solver, re-using or not existing pieces, or use an existing solver. The last option is tempting because we will benefit for years of experience of specialists. Roberto Di Cosmo (Université de Paris VII) led an European project named Mancoosi[24]. The Mancoosi project defined a common dependency description format for Linux distributions: CUDF[25]. CUDF descriptions can be used as input data for many solvers. It would be a good idea to see if our dependency model could be converted to CUDF and thus, be able to use solvers already used to solve Linux dependencies. This work just has been done by the OCAML community with OPAM[26].

References

[BCDL13] Alexandre Bergel, Damien Cassou, Stéphane Ducasse, and Jannik Laval. *Deep Into Pharo*. Square Bracket Associates, 2013.

[DRS+10] Stéphane Ducasse, Lukas Renggli, C. David Shaffer, Rick Zaccone, and Michael Davies. *Dynamic Web Development with Seaside*. Square Bracket Associates, 2010.

[Hen09] Dale Henrich. Metacello, a package management system for smalltalk [software]. https://github.com/dalehenrich/metacello-work, 2009.

[MLW05] Eliot Miranda, David Leibs, and Roel Wuyts. Parcels: a fast and feature-rich binary deployment technology. *Journal of Computer Languages, Systems and Structures*, 31(3-4):165–182, May 2005.

[24] http://www.mancoosi.org/

[25] http://www.mancoosi.org/cudf/

[26] http://opam.ocamlpro.com/

Towards agile cross-platform application development with Smalltalk and Model Driven Engineering

[ab]Glenn Cavarlé [a]Alain Plantec [a]Vincent Ribaud [b]Christophe Touzé

[a]Lab-STICC, UMR CNRS 6285, Université de Bretagne Occidentale, UEB, 20 av. Le Gorgeu, Brest, France
[b]SARL Diazol, 156 rue Jean Jaurès, Brest, France
{glenn.cavarle,alain.plantec,vincent.ribaud}@univ-brest.fr, christophe.touze@diazol.com

Abstract

Nowadays, general public applications or specific information systems must be able to run on mobile platforms as well as on conventional platforms. Because developers have to deal with mobile platform specificities, this constraint significantly lessen the benefits of agile methods and as a consequence impacts on the application development cost. Many research works aim at reducing the development cost. Prototyping as well as automatic code generation have been investigated by the community. In this article, we present Dali, a framework that uses both Smalltalk and Model Driven Engineering. With Dali, an application model can be designed for multiple platforms and interpreted before code generation. An execution platform is modelled as a set of constraints over a context. These constraints can affect the presentation of the Graphical User-Interface (GUI) but also the overall behaviour of an application.

Keywords cross-platform design, agile development, model driven engineering, smalltalk

1. Introduction

Nowadays, general public applications or specific information systems are often able to run on mobile platforms as well as on conventional platforms.

This recent requirement has impacts on the development model, the tools and the development infrastructure. While the cost of purchasing applications decreases, the cost of development increases.

Many research works aim at reducing the development cost. Prototyping is one solution. It can be used to adjust an application before its development for the target platform. In this context, the right capabilities of Smalltalk regarding agile development and prototyping are recognized. Using Smalltak, a prototype can be early tested. However, we are not aware of a Smalltalk infrastructure able to generically deal with different execution platforms for the same application prototype. Moreover, producing the final application to the target platform may involve a significant cost. To overcome this overhead, Model Driven Engineering (MDE) and automatic code generation are often used : stemming from specific models, all or part of the application is automatically generated for the target platform. Produced applications are validated by running them on the target platform. The advantage of this method is to have a single, standard meta-model for all application variants. However, the development cycle is still expensive because the applications have to be tested and validated after code generation.

In this paper, we propose a development method and a framework called Dali based on Smalltalk for early consideration of different execution platforms. With Dali, an application model can be used for several platforms and interpreted before code generation. An execution platform is modelled as a set of constraints over a context. These constraints can affect the presentation of the Graphical User-Interface (GUI) but also the overall behaviour of an application.

The contributions described in this article are :

— we present the Dali framework and an agile development method for early consideration of different execution platforms ;
— we show how an application can be tested before code generation thanks to the interpretation of application models while considering the different execution platforms envisaged for the application.

The remainder of this paper is organized as follows. Chapter 2 explains the problems addressed and existing solutions. Chapter 3 presents our solution called Dali. Chapter 4 presents an illustrative case. Chapter 5 presents Dali meta-models. Chapter 6 explains how Dali is implemented in Pharo. The article ends with relative works and a conclusion.

2. Problems and state of the Art

Our concern is about developing small applications, especially information systems that can be characterized as follows :

1. development is physically centralized on one site ;
2. implementation needs a small number of developers (between 1 and 3) ;
3. targeted system is not critical, i.e. malfunctioning is not likely to endanger people life or health and does not involve significant financial loss to the development company ;
4. the number of screen views is low (between 10 and 20).

These assumptions are in favor of the application of agile methods [TFR05]. Indeed, according to the manifesto for agile software development [BBB+09], some fundamental principles of agile methods are :

— the priority is to meet the needs ; the customer is part of the development team and he operates continuously for validation tasks ;
— to measure the progress of the project regarding the working system and to deliver frequent releases of the system under development ;
— to maintain the simplicity of developments [Hic11].

Considering these principles include notably the following practices [TFR05] :

— development granularity is small ; the code is built iteratively and incrementally and iterations are short ; the expected gain is the early detection and correction of malfunctions ;
— prototypes are developed and tests performed early in the development cycle in relation to the needs expressed by the user ; verification, validation and prototyping activities can be conducted continuously at each iteration.

These practices are consistent with agile development. However, because of the growing importance of mobile platforms, applications shall be developed for desktops and simultaneously adapted to smartphones and tablets.

The idea is similar to that of usage contexts [SeTC99] about the execution platform, the user and the environment. For an application, considering numerous possible contexts involves more complex developments and thus a loss of agility.

2.1 Parallel versions

To support multi-target platforms, for business with limited resources, one direct solution is to maintain different versions, one for each execution platform. Figure 1 depicts such a solution. For each target platform, a part of the application code is written according to the target platform specific library.

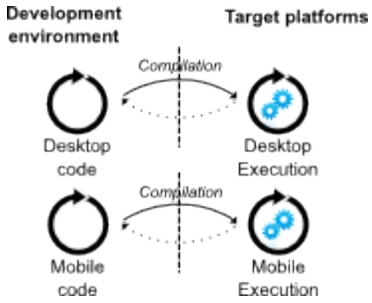


FIGURE 1. Maintaining several versions for the same application

Object-oriented languages can be effectively used to refine abstractions and consider specific contexts such as the features of an execution platform. The solution is to develop a reference version which is then adapted to different platforms envisaged by specialization. Aspect-oriented programming [KLM+97] can provide additional solutions to adapt an application to different contexts.

However, maintaining multiple parallel versions for multiple execution platforms is a very heavy task and makes the continuous validation and tests very difficult. This kind of method can lead to spaghetti code with very negative impact on the evolutivity and on the maintainability of the application code.

2.2 Model-Driven Engineering and code generation

Model-Driven Engineering (MDE) proposes to solve these problems by model transformation and code generation.

As an example, the Model-Driven Architecture (MDA) [Obj00] approach aims at developing a set of models, linked by transformations. These transformations allow to start from a Computation Independent Model (CIM) to a Platform Independent Model (PIM) and finally a final Platform Specific Model (PSM). The PSM represents the concrete implementation of the system. MDE helps to limit specific and non-automated developments for each context : stemming from a cohesive set of models, the different versions can be generated.

The MDE improves the development method because a target platform can be modelled separately. As depicted in Figure 2, from an application model (PIM in MDA) together with a specific platform model, a target platform application model (PSM) is produced by model transformation. Then, parts of the code of the application is automatically generated from it.

The main advantage is that the target platform are specifically modelled and that the same application model can be shared for multiple target platforms.

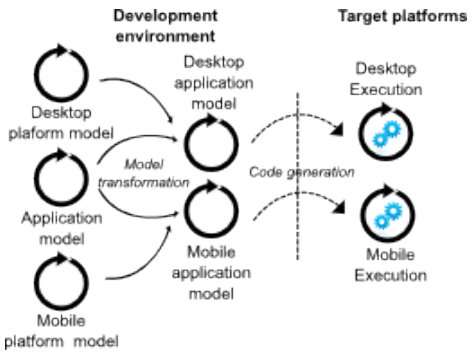


FIGURE 2. The MDE approach

However, depending on the target platform, one must go through a phase of code generation and code compilation before you can execute all or part of the application. These steps are time consuming and the impact of model changes cannot be immediately observed. The causal connection between the models and the generated artefacts is lost because of the code generation step [RFBLO01]. But, to remain agile, a development method must favour short development cycles. Thus, MDE does not help in limiting the loss of agility.

2.3 Emulators

Some development environments provide emulators for target platforms. Emulation relies on the execution of a virtual device supporting the application that will be later deployed. This is, for instance, the solution proposed by the SDK Android [And].

This solution allows to test an application for multiple similar targets without having the physical devices. But, the application execution by the emulator can only occur after the code generation and its compilation. In addition, the application code is specific to one development environment and one kind of emulator. As depicted in Figure 3, this solution implies code generation and deployment steps. In fact, this solution is very similar to the development of parallel versions but with the facility brought by emulators. Because of the code generation step and of the additional deployment steps, the problem of lack of agility is not resolved by emulation. Moreover, emulating can be very slow and time consuming.

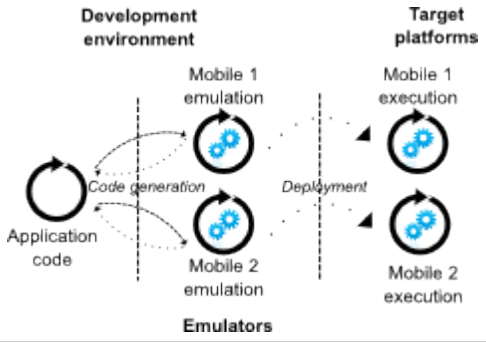


FIGURE 3. Emulation approach

2.4 Common interpreters

A common interpreter can be used for all target execution platforms. Two solutions are possible. The first is depicted by the top part of Figure 4. The interpreter is a separate tool and the application code is the same whatever the platform. This solution is file based. As an example, a web browser can be used as an interpreter of a Javascript program embedded in an HTML page. As another example, one can use Java and its interpreter. The main advantages are that the application code is reusable and web development and deployment is made easier.

Development environments and dynamic languages [Jod10, RG09] promote agile applications development in a uniform and comprehensive way. The second solution is depicted by the bottom part of Figure 4. The interpreter and the development environment are integrated. This is the case of Smalltalk. This solution is image based. As additional advantages, an image based solution favours short development cycle and direct feedbacks. Indeed, with dynamic languages, the causal connection is preserved because any modification of a model can be immediately observed.

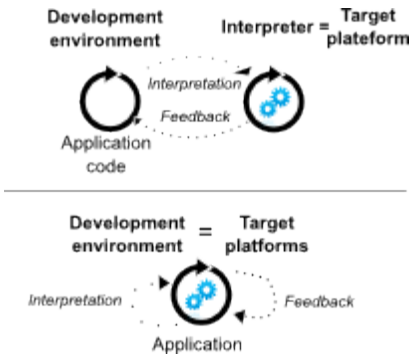


FIGURE 4. Interpretation approach

However, it is necessary to have up-to-date virtual machines for all target platforms. Interpreters can be made available by vendors with specific libraries. This might imply

greater difficulties to maintain code reusability. In addition, the use of interpreters can make it difficult the integration of native widgets and more generally using primitive system. Finally, as for emulation, the interpreting process can be time consuming.

3. Dali solution

To improve the development of applications for different execution platforms while allowing us to have native versions, our solution is blended. We propose to use Smalltalk as a tuning and as a development environment and to use MDE to produce target applications for execution platforms.

Maintaining the causal connection between the models and the executed application can resolve significantly the lack of agility. To maintain it, a solution is to execute an application in and by the development environment.

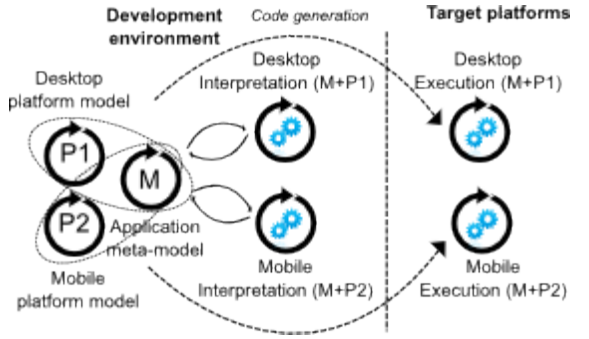


FIGURE 5. The Dali approach

Figure 5 shows the Dali approach. The first goal is to maximize the agility of the development process by the use of a Smalltalk solution to design, prototype and validate an application for several target platform. With Dali, an application model (the business part) must be associated with a specific platform model to be interpreted. The same application model is used for several platforms.

The deployed application can be the Smalltalk one but it might be desirable to directly benefit from native libraries and from fast running. For that purpose, an MDE approach is used. With Dali, after a tuning process, a native application can be automatically generated from the application model together with the chosen target platform model.

The remainder of this paper presents the Dali solution regarding the design on validation steps in Pharo.

4. An illustrative example

Let us take for example the display of a contact list in an address book. Figure 6 shows two possible displays for a contact list. The displayed information as well as the behaviour are different. The left window is adapted to a smartphone and the right window to a desktop computer. On a computer screen, it is possible to view all contact information. On a mobile device, the size is limited. This constraint implies that only the name and the e-mail are presented. A button is added to access the contact details.

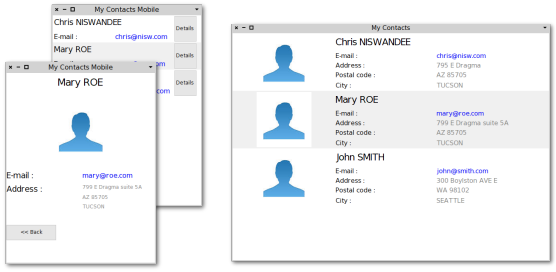


FIGURE 6. Two possible displays of a contact list

We do have the same information system but with two different presentations. We also have a behaviour variation because the mobile phone version requires the use of an additional button.

The solution implemented with Dali let us use the same application model and to express the constraints of the mobile phone. Throughout this article, the presentation of the framework will use this example.

5. The Dali framework

To achieve a satisfactory level of agility in considering specific execution platforms, one solution is to allow the developer to model the various platforms and provide an execution of the application within the development environment. The idea is to converge to a suitable solution before code generation.

The remainder of this section presents the meta-model and the main framework components. We depict features related to the description of business models : specific behaviour, presentation and constraints of execution platforms.

5.1 The Dali meta-model

Dali was inspired by self-descriptive object-oriented meta-models such as EMOF [Gro04] and ECore [SBPM09], especially for the description of the structural aspects of an object. In fact, as in EMOF and ECore, Dali is based on a Classifier (or Class) concept, on Operation and on Property definitions. Dali is also inspired from Magritte [RDK07], from its syntax and from its concept of accessors strategy. Moreover, Dali allows the description of the behaviour and of GUI. Specific requirements related to the target platforms can be used to constrain any Dali description element. The business behaviour is expressed in Smalltalk.

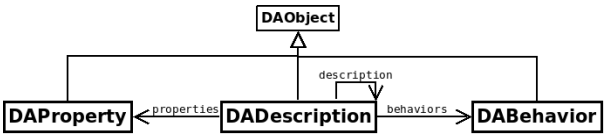


FIGURE 7. Main classes of the Dali meta-model

Figure 7 presents the three main Dali abstractions : *DADescription*, *DAProperty* and *DABehavior*.

A *DADescription* can be related to a Class in EMOF [Gro04]. A *DADescription* handles a set of *DAProperty* and a set of *DABehavior*. A *DADescription* is also a composite : it can contain child descriptions.

A *DAProperty* corresponds to an object instance variable or a Property in EMOF [Gro04]. As a kind of value holder, at runtime, each *DAProperty* stores the value of the related property. A *DAStyle* is a special property used to describe graphic features of a *DAWidget*.

A *DABehavior* might be a business operation (*DAOperation*) as in EMOF [Gro04] but also a more specific behaviour such as a reaction (*DAReaction*), or a binding (*DABinding*).

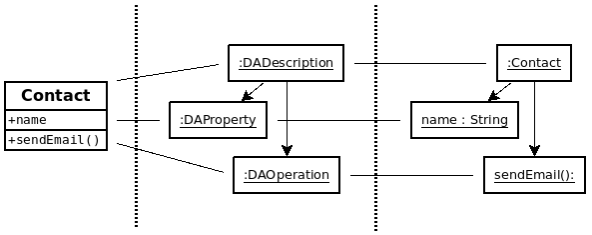


FIGURE 8. Contact class (left), its Dali representation (center) and the simulated instance (right)

Figure 8 shows on the left an UML element for a Contact class with a field $\#name$ and a method $\#sendEmail$. The role of a *DADescription* is twofold : on one side, it describes the structure and the behaviour of the related class, on the other side, it is directly used at runtime to manipulate the value of the properties and to invoke the object behaviour. In the center of Figure 8, is depicted how the Contact class is described with Dali with a *DADescription* instance. The name field is described by a *DAProperty* instance and the $\#sendEmail$ operation is described by a *DAOperation* instance. On the right, is shown a simulated *Contact* instance at runtime. The actual name is stored in the related *DAProperty*. The actual $\#sendEmail$ method is invoked through the related *DAOperation*.

5.2 The behaviour

Dali uses Smalltalk to describe business behaviour. Smalltalk was chosen because the business behaviour can be directly interpreted within the development environment and because it let us the possibility to manipulate the abstract syntax tree (AST) in order to translate the business behaviour to other languages. All business behaviours are manipulated by Dali through instances of *DAOperation*. In fact, a *DAOperation* references the name af the method which implements the related business behaviour.

DABinding and *DAReaction* are more specific. They are implemented as *Observers* [GHJV95] and describe interactions between children of a *DADescription*.

A *DAReaction* specifies an association between a *DAOperation* and an event. For instance, a *DAReaction* can be used to trigger a *DAOperation* as a reaction of an incoming event. Such event can be emitted by a *DAWidget*.

A *DABinding* binds several instances of *DAProperty* together by a source / destination relationship (Data Binding) or bidirectional relationship (Two Way Data Binding). When the source is changed, the destination property is automatically updated (and vice versa if the binding is bidirectional). For instance, a *DABinding* can be used to bind a property of a domain object to a property of a widget.

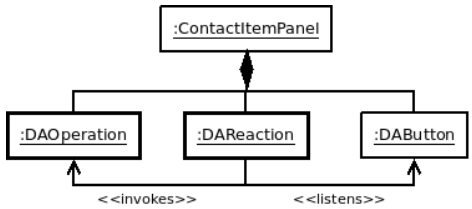


FIGURE 9. Using of a DAReaction

Figure 9 presents an operation triggered as an event reaction. A *DAReaction* listens to an event coming from a source object an is able to trigger an associated operation.

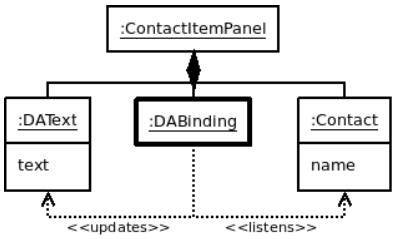


FIGURE 10. Using of a DABinding

Figure 10 depicts how a *DABinding* can be used to automatically chain property updates. When a property value is changed, an event is emitted by the *DAProperty* instance. As a reaction, the *DABinding* updates its target property value.

5.3 The widgets

In Dali, widgets and associated layout are kinds of *DADescription*. Widgets are described as instances of *DAWid-*

get. Each widget has its own set of styles which impacts on the look of the user interface and its own set of managed events which impacts its behaviour. A composition of widgets is specified by a *DALayout* which is owned by a *DAPanel*.

A *DAWidget* is only a logical description which needs to be tied to a native Smalltalk widget at runtime when the application is interpreted. In order to relate a *DAWidget* to a Smalltalk widget, a *DAAdapter* must be specified.

The set of *DAWidget* are defined according to well known standards such as *W3C CSS2* [BLLJ08] for the graphical properties (*DAStyle*) and *W3C DOM* [HKL+13] for events (*DAEvent*). These standards provides the consistency of the API which can cover most of the needs of a GUI description.

5.4 The target platforms modeling

In order to consider execution platforms during the application development, we need models to represent them.

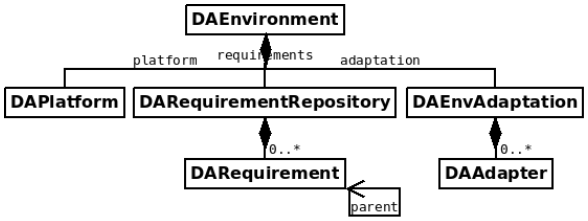


FIGURE 11. The environment composition

As depicted in Figure 11, Dali includes the concept of environment. An environment is reified as a *DAEnvironment* instance. The role of an environment is threefold, it contains the model of the execution platform (*DAPlatform*), the requirements of the application (*DARequirementRepository*) and specifies how the development environment must be customed during the simulation (*DAEnvAdaptation*). Dali makes it possible to simulate and test the same application for several execution platforms. The same set of *DADescription* can be simulated with different actual presentations.

A *DAPlatform* describes a particular target execution platform. A *DARequirementRepository* contains all *DARequirement* used to constrain an application. A *DARequirement* consists in a set of conditions that are checked over a target *DAPlatform*. At runtime, Dali objects structure are set-up according to the related set of *DARequirement*. A *DARequirement* can have a parent requirement. The parent relation is used by Dali to order requirements evaluation, a parent requirement being evaluated before a child one.

Regarding the illustrative example, for the mobile device, a set of *DARequirement* is declared to constrain the size of the panel. Moreover, each row is constraint to present a button instead of showing the complete card data. This kind of constraint is also expressed with a *DARequirement*.

In order to run an application within the development environment, one must declare the actual set of widgets to use (e.g Spec widgets or Morphic widgets). For that purpose, a *DAEnvironment* is set-up with a set of adapters. The purpose of an adapter is to bind a platform widget with a corresponding *DAWidget*. An adapter establishes a mediation between a platform widget and a Dali widget. The mediation consists in providing a concrete look but also in interpreting events and the widget behaviour.

The set of *DAAdapter* is stored within a *DAEnvAdaption* instance. As an example, a *DAEnvAdaption* can be specified for Spec widgets [VRDF12] and another one to bind DAWidget instances directly to Morphic widgets.

6. Dali within Pharo

Dali is implemented in Pharo. The idea is to use Pharo as a classical development environment to design and to early validate applications for multiple target execution platforms but with enhanced agile validation capabilities.

After explanations about the fundamental aspects of Dali regarding DADescription implementation, the reminder of this chapter describes how our illustrative example is implemented in Pharo.

6.1 Fundamentals

All objects manipulated through Dali are instances of a subclass of *DADescription*. Then, a model is made of a set of *DADescription*. It embeds a set of business object descriptions but also the related widget descriptions.

In a Dali description, objects are referenced by a relative identifier (RID). An RID is unique in the scope of a *DADescription* and for all the *DAObject* of a same kind : properties, behaviours and descriptions.

A *DADescription* is made of a list of properties. But, in order to take into account several possible structures and several possible behaviours, the list of properties of a DADescription is not fixed. It depends on the related environment. It means that the actual representation of a concept may differ depending on a given environment.

Each property can be either a data property (e.g. a Contact name) or a behaviour (e.g. the sendMail function). All properties of a *DADescription* are managed as instances of *DAProperty* or of *DABehavior*. It means that a data property is not managed and used through an instance variable but through the corresponding *DAProperty* instance. It means also that a behaviour is not directly coded within a method and, at run-time, is not invoked by a direct message sent but indirectly through a *DABehavior* instanddce.

The methods that implement a particular business object behaviour is still implemented as a method of its class (a subclass of *DADescription*) so that $self$ is preserved at invocation time. Such a method is referenced and indirectly invoked by the corresponding *DAOperation*.

6.2 Configuring an environment

A Dali description structure and behaviour depend on a environment. Thus, an environment must be primarily set-up before instantiating any *DADescription*. The setting of the environment shown in Figure 12 corresponds to the *DAEnvironment* used in the illustrative example.

```
platform := DAPlateform new
      at: #screenWidth put: 300;
      at: #screenHeight put: 400; yourself.

commonRequirement := DARequirement new
      name: #common;
      constraint: [:plfm | true ]; yourself.

smallScreenRequirement := DARequirement new
      name: #smallScreen;
      parent: #common;
      constraint: [:plfm |
            (plfm at: #screenWidth) <= 360 and:
            (plfm at: #screenHeight) <= 480 ]; yourself.

applicationRequirements := DARequirementRepository new
    addRequirement: commonRequirement;
    addRequirement: smallScreenRequirement; yourself.

morphicAdaptation := DAEnvAdaptation new
            adapt: TextMorph
            to: DAText
            with: DAMorphTextAdapter; yourself.

myEnv := DAEnvironment new
   adaptation: morphicAdaptation;
   plateform: platform;
   requirements: applicationRequirements; yourself.

myEnv withinDo: [ContactListWindow new open ].
```

FIGURE 12. Setting and using an environment

An environment associates a *platform*, some *requirements* and an *adaptation* :

— *platform* : A *DAPlateform* stores characteristics of a particular logical platform. All characteristics are stored into a private dictionary so that any characteristic can be freely defined. In Figure 12, an instance of *DAPlateform* is configured to represent a platform with display size characteristics. Regarding our illustrative example, the mobile platform can be described simply by the size of its display.

— *requirements* : A *DARequirement* consists in the specification of a constraint over the platform characteristics. A *DARequirement* is named and is configured with a block. The block implements a constraint over the platform. The constraint block receives the current platform as argument and contains a boolean expression. Finally, all requirements are made available to an environment through a *DARequirementRepository*. Regarding our example :

  — The block of the requirement named $\#common$ always returns $true$, meaning that this requirement is always valid regardless of the platform. Given that the actual list of properties or of behaviours of a *DADescription* is built by evaluating the requirements, this kind of requirement is used to declare that a property or a behaviour is always present in a description.

  — The requirement named $\#smallScreen$ constrains the size of the display. This requirement is used to select the layout to use and which widgets to display.

— *adaptation* : A *DAEnvAdaption* specifies which adapters are used. In our example, we use Morphic as the underlying presentation layer. The adaptation is made of a *DAMorphTextAdapter* which binds *DAText* and *TextMorph*, the Morphic class that is used to edit or display text.

The last line of Figure 12 shows the instantiation of an application configured for a given platform. The environment to be used by the application is made available thanks to the $\#withinDo$: message sent. The application is actually built in the block passed as argument. Internally, the current environment is made available to the application by using the stack context ($thisContext$). Because of the requirement $\#smallScreen$, the resulting application is adapted to a small mobile device as shown in the left part of Figure 6.

6.3 Declaring a business object

The purpose of our illustrative example is to manipulate a list of contacts. A contact is a business object specified by a subclass of *DADescription*. Figure 13 shows the declaration of the *Contact* class.

```
DADescription subclass: #Contact
   instanceVariableNames: ''
   classVariableNames: ''
   category: 'Dali-ContactExample'

Contact>>declareName
   <dali:#common>
   ^DAProperty new
         rid: #name; yourself
```

FIGURE 13. Declaring a business object class

As mentioned in Chapter 6.1, business object properties are not handled through instance variables. Indeed, in Figure 13 the $instanceVariableNames$ string is empty even if a Contact has properties. In fact, any instance variable can be added for internal or private use. But, the business properties must be declared with the help of dedicated methods.

As an example, the Contact *name* property is declared by the $declareName$ method. The property declaration is

based on a specific annotation (*pragma* in Pharo). This annotation is used by Dali to identify the methods to evaluate in order to build the actual list of properties. The property selection is achieved according to the constraints that are declared for the current environment. Such a method is a kind of factory method that returns a configured *DAProperty*.

Thus, with the help of its annotation, the *declareName* method declares that the property named *#name* is added if the requirement named *#common* is met. As shown in Figure 12, the *#common* requirement is always met because its constraint block returns *true*. As a consequence, the *#name* property is always added to the *Contact* description whatever the environment.

```
Contact>>declareLocation
    <dali:#( #smallScreen #gps ) >
    ^DAProperty new
        rid: #location; yourself
```

FIGURE 14. Declaring a property with multiple requirements

In some cases, using a single requirement in the annotation might lead to duplicate requirement code. For instance, suppose that we want to use the GPS service on a smartphone. A field named *location* would be required to store the GPS data. As a consequence, for this property, two requirements must be met : the application runs on the small screen device and the GPS service is available. Figure 14 shows how multiple requirements can be declared with the annotation.

At runtime, a property is used as a value holder. In order to get or set the value of a property, accessors must be implemented. Regarding the *#name* property, we might declare its accessors as shown in Figure 15. Notice that the name of the method is meaningful because it is considered by Dali as the name of the corresponding property. The property instance retrieval is achieved by the *thisProperty* message sent.

```
Contact>>name
    ^self thisProperty value

Contact>>name: aString
    self thisProperty value: aString
```

FIGURE 15. Declaring a property accessors

6.4 Declaring a widget

As explained in Chapter 6, a *DAWidget* is also a *DADescription*. As a consequence, declaring a widget property and its accessors is achieved with the same syntax as for the business objects. A widget is specified through a subclass of *DAWidget*.

Figure 16 shows the declaration of the class *ContactItemPanel* that is the description of the presentation of a contact in the list of contacts shown in Figure 6. A *ContactItemPanel* is specified by a subclass of *DAPanel* which can handle sub-widgets embedded in a layout.

```
DAPanel subclass: #ContactItemPanel
    instanceVariableNames: ''
    classVariableNames: ''
    category: 'Dali-ContactExample'

ContactItemPanel>>declareContact
    <dali:#common>
    ^DAProperty new
            rid:#contact; yourself

ContactItemPanel>>declareNameText
    <dali:#common>
    ^DAText new
            rid:#nameText;
            fontSize: 22 pt; yourself
```

FIGURE 16. Declaring a widget class

As for a property, a sub-widget can be declared by a dedicated method. As shown in Figure 16, the *DAText* named *nameText* is declared by the *declareNameText* method. This method is also used to configure the property *fontSize* of the *DAText* instance.

6.5 Adding a layout

Figure 17 shows the declaration of a row layout referencing widgets of the *ContactItemPanel*. The way a layout is declared with Dali is very near from Spec [VRDF12]. A layout consists simply in a tree of sub-widget references that can be visited without any adaptation and drawing.

```
ContactItemPanel>>declareLayout
    <dali:#common>
    ^DARow new
        child:#contactImage;
        column: [ :c |
            c child: #nameText;
                row: [ :r |
                    r child: #emailLabel;
                        child: #emailText ]]; yourself
```

FIGURE 17. Adding a layout

6.6 Specifying multiple representations

As a solution to multiple representations of the same business object, a Dali description contains all the possible properties regardless of the actual environment. When a description is bound to an environment, the subset of properties that meet the environment requirements is computed.

FIGURE 18. Focus on the presentation of a contact

Figure 18 shows the same element of a contact list but with two different presentations. The top one is for desktop whereas the bottom one is for a mobile platform. This difference is implemented with two possible layouts in the *ContactItemPanel* class. The desktop version is shown in Figure 17. For the mobile version shown in Figure 19, one must declare an additional button and use the *dali* annotation to bind the declarations with the $\#smallSreen$ requirement.

```
1  ContactItemPanel>>declareSmallLayout
2    <dali:#smallScreen>
3    ^DARow new
4            child:#nameText;
5            child:#detailsButton; yourself
6
7  ContactItemPanel>>declareDetailsButton
8    <dali:#smallScreen>
9      ^DAButton new
10           rid:#detailsButton;
11           text: 'Details'; yourself
```

FIGURE 19. Adapting widget with requirements

At instantiation time, the two applicant layouts may be selected. Indeed, the $\#common$ requirement is always met whatever the platform. Dali resolves this selection issue by using the parent relation between requirements. As shown in Figure 12, the $\#smallScreen$ requirement is a child of the $\#common$ requirement. As a consequence, it is always evaluated after the $\#common$ requirement and then, in the context of a mobile platform, only the mobile layout is selected and actually instantiated.

6.7 Adding behaviours

As introduced in Chapter 5.2, a behaviour can be either a *DAOperation* or a *DABinding* or a *DAReaction*. The same declaration syntax is used as for properties, widgets and layouts.

6.7.1 Adding an operation

An operation consists in the declaration of a method to invoke. The name of the method serves as a key to lookup the actual method. Figure 20 shows the declaration of a *DAOperation* and the associated method. At runtime, this operation invocation results in the $\#openDetails$ message sent to the receiver.

```
1  ContactItemPanel>>declareOpenDetails
2    <dali:#common>
3    ^DAOperation new
4            rid:#openDetails; yourself
5
6  ContactItemPanel>>openDetails
7        ContactDetailsWindow new
8            contact: self contact;
9            open
```

FIGURE 20. Declaring an operation

6.7.2 Adding a binding

Figure 21 shows the declaration of a *DABinding*. The *DABinding* establishes a link between the property $\#name$ of the Contact and the property $\#text$ of the child widget $\#nameText$. Thanks to this binding, when the contact name is changed, the $\#text$ property of the widget is automatically updated.

```
1  ContactItemPanel>>declareNameBinding
2    <dali:#common>
3    ^DABinding new
4            rid:#nameBinding;
5            srcAccessor:
6                ((#contact asDaliAccessor)
7                    withNext: #name asDaliAccessor);
8            destAccessor:
9                ( (#nameText asDaliAccessor)
10                   withNext: #text asDaliAccessor);
11           yourself
```

FIGURE 21. Declaring a binding

A binding relies on accessors for the source and the target properties. Figure 22 shows the hierarchy of accessors implemented as a Decorator. An accessor can be straightforward or chained. This feature is borrowed from Magritte [Ren06].

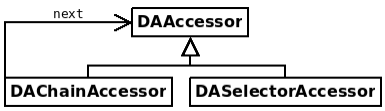


FIGURE 22. Hierarchy of accessors in Dali

6.7.3 Adding a reaction

A reaction consists in the declaration of an association between an event and an operation. Figure 23 shows the declaration of a *DAReaction* which listens the event *DAClick* fired by the button named $\#detailsButton$. When the button is clicked, the operation named $\#openDetails$ is invoked.

```
ContactItemPanel>>declareButtonReaction
    <dali:#smallScreen>
    ^DAReaction new
            rid:#buttonReaction;
            event: DAClick;
            senderAccessor: #detailsButton asDaliAccessor;
            operationRid: #openDetails; yourself
```

FIGURE 23. Declaring a reaction

6.8 More on styles and events

The public protocol of a widget consists mainly in implementing its styles and managing events.

Dali is a modelling layer that must be adapted to specific environments for graphic rendering. To remain adaptable to any underlying framework, Dali does not provide its own underlying graphical implementation and widget protocols are not defined according to a specific one (Morphic, Spec,...). Instead, the goal is to rely on normalized protocols. For that purpose, we have chosen to be compliant with two main standards : W3C CSS2 [BLLJ08] for styles and W3C DOM [HKL+13] for events.

6.8.1 Styles

Styles management can be confusing. Each style can be used by several widgets but sharing of styles can not rely on widget hierarchy definition. As an example, button and text are represented in Dali by *DAButton* and *DAText*. Their common ancestor is *DAWidget*. These two widget kinds can manage a text. Then, these widget kinds must implement text styles management. Unfortunately, the text styles management is not available at the level of *DAWidget*. Thus, we need a clear way to define styles and to allow their sharing independently of the widget hierarchy. For that purpose, as shown in Figure 24, Dali uses traits to declare available styles.

```
DAWidget subclass: #DARectangle
    uses: DATWithBackground + DATWithBorder
    ...
DARectangle subclass: #DAButton
    uses: DATWithText
    ...
DAWidget subclass: #DAText
    uses: DATWithText + DATWithBgColor
    ...
```

FIGURE 24. The text and the button widgets sharing the same text style definition

A style trait covers the definition of a standard subset of CSS2 [BLLJ08] styles. As shown in Figure 25, in such a trait, each style implementation is made of three methods : the first is for the style instantiation and the two others are for the style value accessing. As an example, The *DATWithText* trait shown in Figure 25 is compliant with subset of CSS about text specification.

```
DATWithText>>declareFontStyle
    <dali>
    ^DAStyle new rid: #fontStyle;
        value: DAFontStyle normal; yourself

DATWithText>>fontStyle
    ^self thisProperty value

DATWithText>>fontStyle: aFontStyle
    ^self thisProperty value: aFontStyle
```

FIGURE 25. DATWithFont trait snippet

6.8.2 Events

The DOM [HKL+13] specification define different kind of events that can be fired by graphical elements. Dali provide the *DAWidgetEvent* hierarchy which is compliant with this specification. To take advantage of announcement mechanism in Pharo, *DAWidgetEvent* is a subclass of *Announcement*. Once more, traits are used to make available the event protocols at the widget level. The implementation of an event, in such a trait, is made of two declarations of a *DAOperation* with their related methods. The first method fires a specific event, the second method declare a reaction between this event and an operation. Figure 26 shows the first one.

```
DATWithClickEvent>>declareFireClickEvent
    <dali>
    ^DAOperation new rid:#fireClickEvent; yourself

DATWithClickEvent>>fireClickEvent
    self announcer announce: (DAClick target: self)
```

FIGURE 26. DATWithClickEvent trait snippet

6.9 Visiting a model

As a development environment, Dali permits early validation of applications through their direct interpretation. But the ultimate goal is to produce an adapted version of an application for a specific execution platform. The primary requirement for such a goal is to be able to visit a description. Moreover, a description model must be visitable even it is not adapted.

To visit a model instance, an environment must be configured as described in Chapter 6.2. This environment represents the target platform. In its current state, Dali allows such visits as presented in Figure 27. This example shows a visit invocation in order to generate a CSS representation. The resulting CSS is shown in Figure 28.

```
myEnv whithinDo: [
    DACssExporter new
        visit: (ContactListWindow new)]
```

FIGURE 27. Exporting CSS from a widget

```
div.contact-window {
      width: 300 px; height: 400 px;
      color: #000000; background-color: #FFFFFF;
}
 div.contact-item-panel{
      width: 100%; height: 60 px;
      color: #000000; background-color: #FFFFFF;
}
span.contact-item-panel .contact-name { font-size : 22 pt;}
span.contact-item-panel .contact-email { color : #0000F1;}
```

FIGURE 28. Generated CSS snippet

7. Related Work

AppliDE [QDDD11] is a software framework that is based on Software Product Lines (SPL) [PBL05]. SPL lets the user define characteristics diagrams for each products family also called features. AppliDE uses SPL in association with MDE to construct a single model of products family. Features are used to specify the variability between different products. This model is used to automate the derivation of applications for several target platforms. AppliDE describes the behaviour by high-level services (GPS, mailing,...) and manipulates them as features. The business behaviour is implemented at the target execution platform level. The generated code is linked with the business behaviour after code generation. AppliDE uses a static builder for GUI aspects of an application. AppliDE is code generation based. It does not provide any interpreting facility.

CAPucine [Par11] uses also SPLs but with Aspect-oriented programming to consider context variations at runtime. It is based on a variability model to define product families and their variability points. An aspect model is used at design time to generate code and is used at runtime for reconfigure an application according to the events fired within the execution context. CAPucine is also based on code generation. The behaviour is represented at a high abstraction level, as components or services. Compared to AppliDE, CAPucine does not include any elements to automate the development of the GUI.

SPL features are similar to the Dali notion of requirements. But, the composition of an application is made at high level of abstraction (service level). Whereas, in Dali, a composition of an application is made at the class level.

The works around the plasticity [Cal10] and the multimodality [CCT+03] of GUI take advantage of MDA [Obj00] architecture to abstract and generalize concepts without specialize an application directly to some context. But to allows reconfiguration at runtime, the target platforms must include a specific framework. The Comets [CDCD05] approach brings widgets which can react at runtime to adapt their presentation according to the variation of the runtime context. These Comets are implemented in each target platform and their models are used at design time to specify GUI composition.

In Dali, a platform is modelled at design time and is made to constrain objects structure. But, Dali does not provide any dynamic reconfiguration mechanism after code generation. Thus, the meta-model is not represented at runtime in the target platform.

Magritte [RDK07] is a framework to describe properties of a domain object. The properties description consist in a separate meta-model that can be manipulated apart. It is auto-descriptive. Magritte uses mementos to cache model state and manage model changes as transactions. These mementos can be also used as a kind of value holder.

Dali uses also a meta-model to describe object properties. In Dali, a description is directly used as a business object : its properties serve as value holders and for the invocation of the behaviour.

Spec [VRDF12] is inspired by the VisualWorks UI builder and is implemented in the Pharo environment. It is used to specify and build GUI in Pharo. Spec uses a logical widget model with a specific API and an adaptation layer to bind its widget model to Morphic widgets. The widget composition is implemented with layouts relying on the command pattern. Each layout element exists as a command that is evaluated when the layout is adapted to Morphic. No visiting mechanism is provided for the widget hierarchy.

A part of Dali is dedicated to GUI description. As in Spec, it implements widget composition, layouts and an adaptation layer. Dali widgets are auto-descriptive and can be reconfigured at design time according to platform constraints. Moreover, Dali style and events are compliant with W3C [BLLJ08, HKL+13].

8. Conclusion and further work

This paper deals with the problem of agile application development for multiple execution platforms. Nowadays, application vendors must provide multi-platform products. Often, existing desktop applications have to be adapted or rewritten to be compliant with mobile platform. This constraint impacts the cost of application development even with agile technologies.

This paper presents a solution named Dali which benefits from Smalltalk together with the Model Driven Engineering to allow multi-platform application design. With Dali, an application can dynamically take into account a platform description and can be interpreted to allow agile design and validation. When the application is mature enough, the idea is to finally generate a native target application.

In its current state, Dali provides a framework that can be used to design desktop as well as mobile simple applications.

The current set of available widgets and related adapters remain to be enriched. A Dali model can be visited. It provides the mean to actually generate target platform code. But, generating a real application remains to be experimented.

At the design level, two perspectives are under consideration. The first one concerns the property definitions that could benefit from Slots [VBLN11]. Indeed, a property description is a specific instance variable with particular management rules.

In Dali, target platform description consists in a set of properties. A pragma based mechanism is used to select the actual set of business object properties according to specific environment requirements. The pragma mechanism implies a class based definition of all possible properties. This part of the framework could benefit from an Aspect oriented mechanism. Indeed, the definition of properties available in a given environment could be designed with the help of Aspects. We plan to explore the use of PHANtom [Fab12] for that purpose.

Références

[And] Android sdk. http ://developer.android.com/.

[BBB+09] Kent Beck, Mike Beedle, Arie Van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, and Jeff Sutherland. Manifesto for agile software development. http ://agilemanifesto.org/, 2009.

[BLLJ08] Bert Bos, Håkon Wium Lie, Chris Lilley, and Ian Jacobs. Cascading style sheets, level 2 (CSS2) specification. W3C recommendation, W3C, April 2008. http ://www.w3.org/TR/2008/REC-CSS2-20080411/.

[Cal10] Gaelle Calvary. Plasticité des interfaces homme-machine : Rétrospective et perspectives. pages 3–4, Marseille, France, 2010.

[CCT+03] Gaëlle Calvary, Joëlle Coutaz, David Thevenin, Quentin Limbourg, Laurent Bouillon, and Jean Vanderdonckt. A unifying reference framework for multi-target user interfaces. *INTERACTING WITH COMPUTERS*, 15 :289–308, 2003.

[CDCD05] Gaëlle Calvary, O. Daassi, Joëlle Coutaz, and A. Demeure. Des widgets aux comets pour la plasticité des systèmes interactifs. *Revue d'Interaction Homme-Machine (RIHM)*, Volume 6, n°1, ISSN 1289-2963 :33–53, 2005.

[Fab12] Johan Fabry. Phantom : An aspect language for pharo smalltalk. In *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development Companion*, AOSD Companion '12, pages 31–32, New York, NY, USA, 2012. ACM.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Gro04] Object Management Group. Meta object facility (mof) 2.0 core final adopted specification. 2004.

[Hic11] Rich Hickey. Simple made easy. StrangeLoop 2011 conference talk, http ://www.infoq.com/presentations/Simple-Made-Easy/, Oct 2011.

[HKL+13] Philippe Le Hégaret, Gary Kacmarcik, Travis Leithead, Tom Pixley, Björn Höhrmann, Doug Schepers, and Jacob Rossi. Document object model (DOM) level 3 events specification. W3C working draft, W3C, November 2013. http ://www.w3.org/TR/2013/WD-DOM-Level-3-Events-20131105/.

[Jod10] André Jodoin. *Un environnement dynamique de développement (EDD) pour le prototypage rapide d'interfaces graphiques*. PhD thesis, Université de Montréal, 2010.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. *Proceedings of ECOOP '97, LNCS, Springer-Verlag*, 1241 :220—-242, June 1997.

[Obj00] Object Management Group. Model Driven Architecture (MDA), 2000.

[Par11] Carlos Parra. *Towards Dynamic Software Product Lines : Unifying Design and Runtime Adaptations*. These, Université des Sciences et Technologie de Lille - Lille I, March 2011.

[PBL05] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering : Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[QDDD11] Clément Quinton, Christophe Demarey, Nicolas Dolet, and Laurence Duchien. AppliDE : modélisation et génération d'applications pour smartphones. In *Journées sur l'Ingénierie Dirigée par les Modèles (IDM'11)*, pages 41–45, Lille, France, June 2011.

[RDK07] Lukas Renggli, Stéphane Ducasse, and Adrian Kuhn. Magritte - a meta-driven approach to empower developers and end users. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *Model Driven Engineering Languages and Systems, 10th International Conference, MoDELS 2007, Nashville, USA, September 30 - October 5, 2007, Proceedings*, volume 4735 of *Lecture Notes in Computer Science*, pages 106–120. Springer, 2007.

[Ren06] Lukas Renggli. Magritte — meta-described web application development. Master's thesis, University of Bern, jun 2006.

[RFBLO01] Dirk Riehle, Steven Fraleigh, Dirk Bucka-Lassen, and Nosa Omorogbe. The architecture of a uml virtual machine. *SIGPLAN Not.*, 36(11) :327–341, October 2001.

[RG09] Lukas Renggli and Tudor Gîrba. Why smalltalk wins the host languages shootout. In *IN : PRO-*

CEEDINGS OF INTERNATIONAL WORKSHOP ON SMALLTALK TECHNOLOGIES (IWST 2009), ACM DIGITAL LIBRARY, 2009.

[SBPM09] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF : Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.

[SeTC99] Angela Sasse, Chris Johnson (editors, David Thevenin, and Joelle Coutaz. Plasticity of user interfaces : Framework and research agenda, 1999.

[TFR05] Daniel E. Turk, Robert B. France, and Bernhard Rumpe. Assumptions underlying agile software-development processes. *J. Database Manag.*, 16(4) :62–87, 2005.

[VBLN11] Toon Verwaest, Camillo Bruni, Mircea Lungu, and Oscar Nierstrasz. Flexible Object Layouts : enabling lightweight language extensions by intercepting slot access. In *Proceedings of 26th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '11)*, Portland, États-Unis, November 2011.

[VRDF12] Benjamin Van Ryseghem, Stéphane Ducasse, and Johan Fabry. Spec : A Framework for the Specification and Reuse of UIs and their Models. In *Proceedings of ESUG International Workshop on Smalltalk Technologies (IWST 2012)*, Gent, Belgique, August 2012.

Tracking dependencies between code changes: An incremental approach

Lucas A. Godoy
INRIA Lille-Nord Europe UBA
lucas.ariel.godoy@inria.fr

Damien Cassou Stephane Ducasse
INRIA Lille-Nord Europe
damien.cassou@inria.fr stephane.ducasse@inria.fr

Abstract

Merging a change often leads to the question of knowing what are the dependencies to other changes that should be merged too to obtain a working system. This question also arises with code history trackers – Code history trackers are tools that react to what the developer do by creating first-class objects that represent the change made to the system. In this paper, we evaluate the capacity of different code history trackers to represent, also as first-class objects, the dependencies between those changes. We also present a representation for dependencies that works with the event model of *Epicea*, a fine-grained and incremental code history tracker.

Keywords change propagation, IDE, history, dependency analysis, software evolution

1. Introduction

Software systems evolve in response to change in their functional requirements. These changes made through time to the source code of software systems is what we call their code history. We can keep track of this evolution process through the usage of Version Control Systems (VCSs) such as Git[1].

Since software engineering is part of software evolution [RL07], a development environment that represents changes as first-class entities that can be referenced, queried and passed along in a program [EVC+07] is fundamental for a *change-oriented* engineering approach. This cannot be accomplished using the mainstream VCSs in use today for the following reasons:

- The semantic information of the changes made to the system is scattered in a large amount of text, so tracking entities involves parsing several versions of the entire system.
- Several independent fixes and features can be introduced in one single commit, making it hard to differentiate them.
- The time information of each change is reduced to the time when each commit is performed, so all information about the exact sequence of changes which led to these differences is lost.

To minimize the effort for sharing and merging code through a VCS, some best practices have been established:

- Commit small, related, self-contained change sets. This is what is usually known as an *atomic commit*[2].
- Usage of a descriptive commit message.
- Commit regularly.

Following these best practices requires a lot of discipline. As a result, committing unrelated changes happens regularly in software development. This means that either the tools are used do not allow to follow the best practices or the effort to follow the aforementioned best practices is too high for developers.

To reduce this effort, a new generation of tools (that we call *code history trackers*) was born. These tools are conceptually *event-based*: they react to what the developers do by creating first-class objects that represent the changes made to the system. Remarkably Smalltalk change tracking systems (ChangeSorter) is one the elder code history trackers and it predates mainstream versioning systems.

However, we consider that none of the current code history trackers has a minimal set of desired features to reduce the effort required to do an atomic commit. The most important of these features is the ability to detect dependencies between changes made to the system, or what we call *dependency tracking*.

In general, the *re-assembly of changes* has been historically supported through a feature called *cherry-picking*. The support for cherry picking enables programmers to extract incremental improvements that are spread over a set of many changes. Consider for example that a task has involved a refactoring that the programmer must manage and share as a separate improvement. Programmers can first have a look at the list of all versions to identify both the individual changes that constitute the refactoring and the version from which the main task started.

Over time, it becomes increasingly difficult and tedious for a developer to determine whether a change from another branch or fork can benefit the system, which makes it difficult and time consuming. This difficulty is emphasized by the lack of support for the analysis of dependency between changes. Indeed it is rare that a change happens in isolation.

There is a need for tools that can detect dependencies automatically, so the programmer doesn't need to remember these dependencies or to identify and select them manually.

The contributions of this paper are:

- An evaluation of current history tracking tools for Smalltalk and how they facilitate the dependency tracking to reduce the human effort needed to follow the described best practices.
- The definition of one model and the building of a dependency tracking mechanism on top of it, focusing on simplicity, to assist the programmer in the process of re-assembling changes. Given the dynamic nature of Smalltalk, the approach is not completely accurate for message sends. And it is not fully automatic, since the developer has always the chance to edit the suggestions of the tool.

[1] `http://git-scm.com/`

[2] `http://en.wikipedia.org/wiki/Atomic_commit`

Table 1. Evaluation summary

System	First-class objects	Incremental	Dependencies	Refactors	Exploration
ChangeSet	Partial	✓	✗	✗	ChangeSorter
Ring	✓	✗	✓	✗	Jet
Epicea	✓	✓	✗	✓	Log Browser
CoExist	✓	✓	✗	✗	Version Bar

2. Related work

In this section we describe three existing tools for code history tracking and we evaluate how they assist the developer to facilitate the best practices listed earlier.

2.1 Evaluation criteria

To evaluate the existing tools we consider the following questions:

- Are changes modelled as first-class objects?
- *Is it incremental?* Is it possible to analyze a single change or does it need to create a full history log? Incrementality makes the semantic representation of the model easier to mantain.
- Are dependencies between changes modelled?
- Are high-level refactorings modelled?
- Does the solution provides a flexible way to explore the list of changes?

2.2 The Smalltalk ChangeSorter/ChangeSet

The traditional Smalltalk ChangeSet log is a reliable mechanism to log the source modifications immediately after any editing operation happens on an image [Gol84]. It may be used as a recovery tool by backtracing to the most recent non/erroneous state of the image and reapplying changes listed by the ChangeList tool.

However, the changes are written to a log file as executable statements and only classes and methods are modelled as first-class objects. Additions and removals of attributes can only be detected by comparing different versions of the program. These records have no information about high-level changes as refactorings and mix source management with the events that make the system evolve from one state to another. As a result, not all events can be recorded, the granularity of the events is too coarse and the exploration of the change list made with the ChangeSorter is cumbersome and error-prone.

Considering these limited the representation of changes, is no surprise that the model does not include a representation for dependencies between changes.

2.3 Ring

Ring [UGDD12] is a unified source code meta-model that:

- Has a common API with the Runtime and Structural Smalltalk model.
- Represents every program entity as a first-class object. Unlike the standard Smalltalk model, it can represent variables as objects instead of strings.
- Serves as the underlying meta-model for the history and change meta-models.

The history meta-model, called *RingH*, models source code entities such as packages, classes, methods and atributes as well as the relationships between such entities such as class inheritance, method call, class reference and attribute access. The history models are extracted from the source code history contained within versioning repositories.

RingC is the change and dependency meta-model, which uses the information contained within the RingH model and creates sets of changes (instances of class RGChange) for each snapshot retrieved from the repository. These sets are called *deltas*. When there is a dependency between two RGChange objects, the RingC model creates a RGChangeDependency that represents it. This dependency can happen within changes in the same delta or between changes belonging to different deltas. If a reference to a non-existing object is introduced in a change, that change has an external dependency that is modelled with a stub class.

Even when the object model improves the granularity of the events recorded, the Ring approach is still unable to detect the most high-level events (i.e: refactorings).

Jet [UG12] is a semi-automated tool built on top of RingC that offers a characterization of the changes and dependencies within a stream of changes. It is not incremental, since it creates a full history log by extracting information from repositories instead of reacting in real-time to the changes made by the developer. Because of this, the process of importing the repository data to generate the history and then extract the changes and their dependencies can be time consuming for big projects.

2.4 CoExist

CoExist [STCH12] is a tool for Squeak/Smalltalk that relies on the idea of continuous versioning: any change made to the system triggers the creation of a new version storing the change as well as a complete snapshot of the current system. Unlike a traditional VCS that store the source code changes in separate files, these snapshots are internal data structures that store the state of the system in a particular point in time.

The user can go back and forth between versions using the *Version Bar* and create additional working environments to inspect, modify and debug versions of interest. The system also allows to run tests continuously, to collect results for every individual version and to run a potentially new test on previously created program versions.

Despite these features not found in the other tools, CoExist is not free of limitations. Some classes cannot be versioned and switching between versions requires a restart of the application under development. It also lacks support for direct references to class objects and its model does not include dependencies between changes.

2.5 Epicea

Epicea [DCD13][3] is a code history tracker built on top of the Ring core that represents the changes in entities with events. Each one of these events contains two snapshots representing how the entity was before and after the change. This capacity of reacting to the events as they happen provides the exact sequence of changes that led to the differences between each pair of snapshots. It is also easier to maintain a semantic representation of the model, requiring code parsing only at the method level.

The event model has some trade-offs between accuracy and simplicity. For example every time Epicea detects a change in a

[3] `http://smalltalkhub.com/#!/~MartinDias/Epicea`

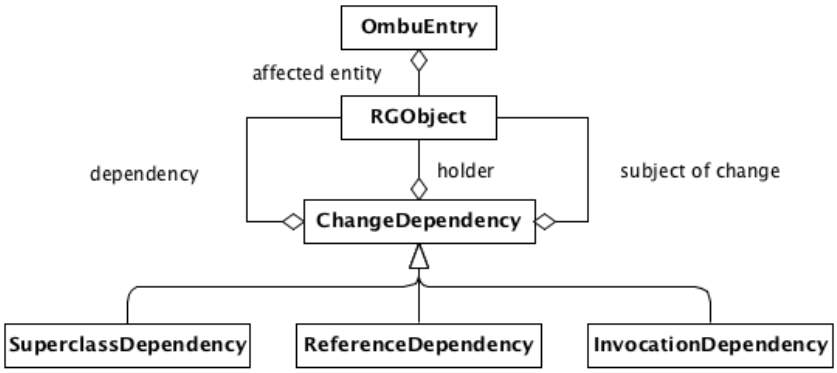


Figure 1. Object model

class, it is unable to distinguish between an addition or removal of an instance variable and the addition or removal of a class variable. This is not a major drawback for its current features, but it is something that will have to be considered if we want to add dependency tracking to its feature set.

Epicea writes each event immediately to disk using one Ombu file per session instead of a single ChangeSet file, making easier the recovery of the exact sequence of changes that originated the differences between the snapshots of the affected entity. It also can export the log entries to a ChangeSet file (only for events supported by the ChangeSet format).

Unlike the standard ChangeSet model and the complete Ring model, Epicea events can represent high-level refactorings. This simple event-based model replaces the RingH layer of the Ring ecosystem but there is no model to represent dependencies between the changes triggered by those events.

Also the Log Browser makes easy to go back and forward in time using the events logged, leading to an easier exploratory development.

3. Additions to the Epicea object model

Our objective is to create an object model to represent dependencies between the existing change model of Epicea. In this section we define what a dependency is, how to represent a dependency with an object and how to extract the dependencies from each changed entity in the system.

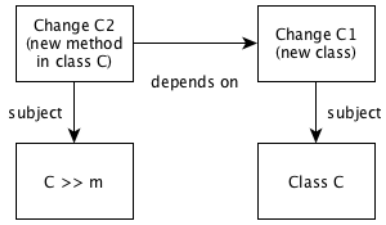


Figure 2. Dependency in a method creation

A change is always applied to a *subject*. *Creational changes* are changes which have as subject a new entity that they produce. In this case, a change `c1` is said to depend on a change `c2` if that is the creational change of the subject of `c1` [Figure 2]. For example, methods can only be added to existing classes.

Also, the source code of `m` can contain references to other entities and messages sent [Figure 3]. The entities referenced in the source code of `m` must exist to ensure its compilation and proper execution. Because of this, we need to parse the source code associated to every change.

Therefore, our dependency object [Figure 1] will be composed of three references to entities:

1. The subject of change or entity to be modified.
2. Optionally, a class holder. This is the class that holds the subject of change. It will be nil for classes, since they don't need a holder.
3. Optionally, a set of dependencies extracted from the source code. If the event is a class addition or any entity removal, it will be empty.

This set of dependencies is generated from the source code of methods. In the next subsections we explore the different dependencies that we can find.

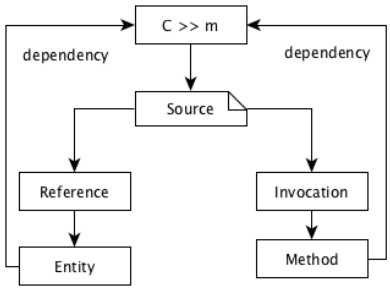


Figure 3. Dependencies extracted from a method

3.1 Types of dependencies

We have three types of dependencies between entity changes. In all cases, parsing the code associated to the entity is needed.

- *Class hierarchy dependencies*: for each change in a class, which can be a change in a method or in the class definition, the superclass must exist. The same happens when self is called from the code of the method that is the subject of change.

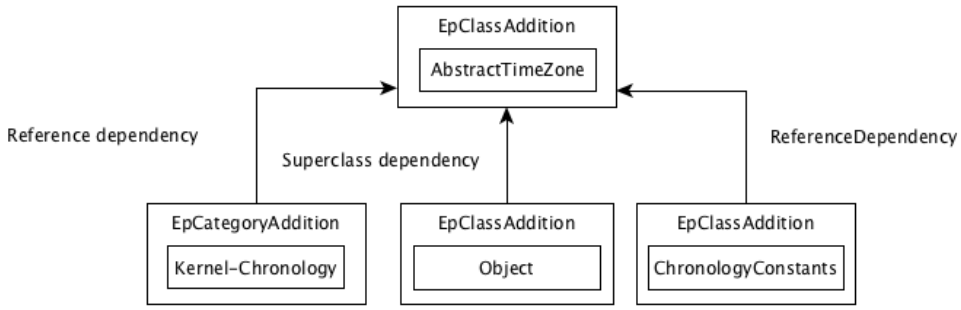


Figure 4. Dependencies of a class addition

- *Reference dependencies*: they are references to temporary, instance and class variables in the source code of any method. Also references to classes.
- *Message sends*: messages sent in the source code of any method or expression evaluation. Since Smalltalk is dynamically-typed, in absence of type information and presence of polymorphism, there is a need to provide very fine-grained information about messages sent to find dependencies in an accurate way.

It may happen that a dependency for a change is located in a different package. We call this an external dependency. And if the dependency doesn't exist in the system, we call it a missing dependency. Both cases will have their first-class object in our model.

```
1 Object subclass: #AbstractTimeZone
2        instanceVariableNames: ''
3        classVariableNames: ''
4        poolDictionaries: 'ChronologyConstants'
5        category: 'Kernel-Chronology'
```

Listing 1. Class definition example

Listing 1 shows the code of the class AbstractTimeZone. This class inherits from Object, uses the pool dictionary ChronologyConstants and is located in the category Kernel-Chronology. So we can extract 3 dependencies from this definition [Figure 4]:

1. The class Object must be defined.
2. The shared pool ChronologyConstants must be defined. This also means that the class SharedPool must be defined.
3. The package Kernel-Chronology must be defined.

```
1 Trait named: #TClass
2        uses: TBehaviorCategorization
3        category: 'Traits-Kernel-Traits'
```

Listing 2. Trait definition example

Trait definitions are similar. The dependencies in Listing 2 are the Trait class, the TBehaviorCategorization trait and the category.

3.1.1 Message sends

If a message is sent inside the code of a method, we can look for the methods that potentially will receive the call (i.e., dynamic dispatch). This is what we know as a *candidate set* [DAB+11]. Since candidate sets can contain false positives, we categorize message sends as follows:

- Messages sent to self: all candidates for the call need to be in the hierarchy tree of the class in which the method is defined. This case can lead to false positives when the method is declared in many classes that belong to same hierarchy.
- Messages sent to super: this corresponds to the super calls within a method, which is bound statically. So it must be defined in a direct or indirect superclass in which the method is defined.
- Messages sent to classes: The receiver of this message is a class reference.
- Unknown sends: the call of the receiver is unknown, so the candidate set consists of all methods with the given selector. This case can lead to false positives.

```
1 AbstractTimeZone >> printOn: aStream
2        super printOn: aStream.
3        aStream
4                nextPut: $(;
5                nextPutAll: self abbreviation;
6                nextPut: $).
```

Listing 3. Method definition example

Listing 3 shows the code of method printOn: from the class AbstractTimeZone. We can extract many dependencies from this change [Figure 5]:

1. First of all, we need the class AbstractTimeZone to add the method.
2. At line 2, we have the message printOn: sent to super, so this depends on Object ≫ printOn: or ProtoObject ≫ printOn: according to the class hierarchy of AbstractTimeZone.
3. At line 5, we have a message send to self. So this depends on a method called abbreviation that can be on any member of the current class hierarchy.
4. Starting at line 3, we have many messages sents to a parameter. Since we don't know to which class the parameter belongs, this is an unknown invocation. It's candidate set are all the methods called nextPut: and nextPutAll:.

Listing 4 shows the code of AnnouncementSpy ≫ buildList. Since we are sending the message new to the class PluggableListMorph, we are sure that this class and its method must be defined. We could assume the result of this message is an instance of PluggableListMorph, but since we cannot be sure that new returns an

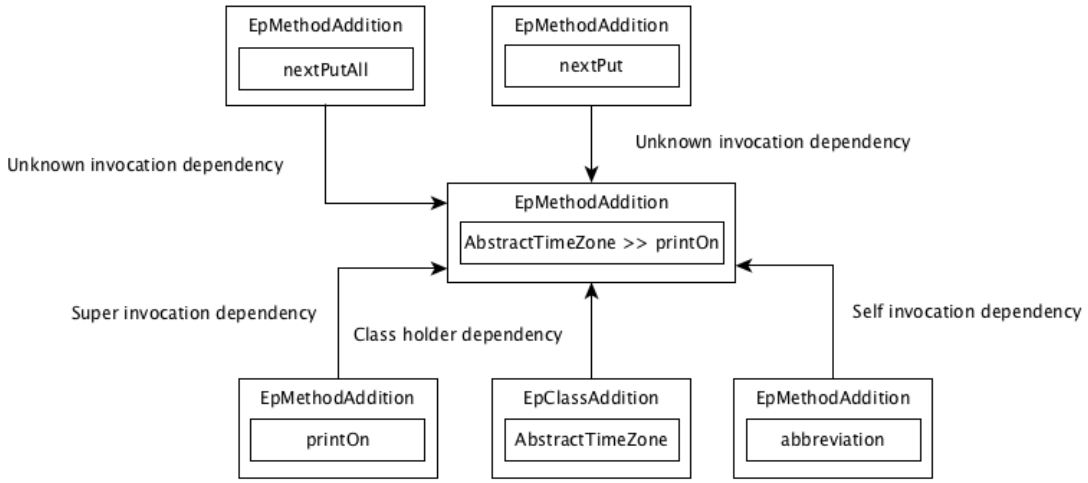


Figure 5. Dependencies of a method addition

instance of the class, we're forced to look for all the implementors of the message.

```
buildList
        ^ (PluggableListMorph new)
                on: self
                list: #announcements
                selected: #index
                changeSelected: #index:
                menu: #buildMenu:
                keystroke: nil.
```

Listing 4. Example of a message sent to a class

3.2 Unknown message sends with self

Let's suppose we added the method addAll: to the class Collection [Listing 5]:

```
addAll: aCollection
        aCollection do: [:each | self add: each].
        ^ aCollection
```

Listing 5. Unknown message sends with self

Among others, we have a dependency with add:. It's code is shown on Listing 6.

```
add: newObject
        self subclassResponsibility
```

Listing 6. Role of subclassResponsibility

This one has a dependency with subclassResponsibility. But this is not enough to make addAll: work. We should include all the add: messages in the Collection hierarchy. This is a case where a dependency found in a message send to self can have false positives.

3.2.1 Reference dependencies

Let's illustrate how to handle variable references by looking at the code of this method in the class OrderedIdentityDictionary [Listing 7].

We have messages sent to self and super, that we already covered. We have an unknown send for the message key, that has more 13 implementors in a standard Pharo 3.0 [4] image. We also have an unknown sned for ifFlase:, but in this case there are only 3 implementors restricted in the Boolean class hierarchy.

```
add: anAssociation
        (self includesKey: anAssociation key)
                ifFalse: [ keys add: anAssociation key ].
        ^ super add: anAssociation
```

Listing 7. Reference dependency example

We also had a reference to a variable called keys. There is no temporary variable declared in the source code of the method, so it must be an instance or class variable. We said in Section 2.3 that Epicea cannot distinguish between diferent kinds of variable changes in a class definition. This means that we'll have a RGClassDefinition entity that contains a class or instance variable as a dependency. Now a question is raised: which class definition is the one that contains this variable?

```
atRandom: aGenerator
        | rand index |

        self emptyCheck.
        rand := aGenerator nextInt: self size.
        index := 1.
        self do: [:each |
                index = rand ifTrue: [^each].
                index := index + 1].
        ^ self errorEmptyCollection
```

Listing 8. Local variable reference

The answer is that this variable should be defined in the last event containing a class definition for A, otherwise the code would not compile (unless the compiler decides to skip compiling for some reason). The worst case would be when the class was created before the installation of Epicea. if this happens, it will scan the full log only to find that the dependency is missing.

[4] http://pharo.org/

In listing 8 we have an example of temporary references, extracted from the method atRandom: of the class Collection.

Since rand and index are defined in the same method, there is no dependency. We could think that there is a dependency with the method itself, but compilation is not possible without the declaration of these two variables.

3.3 Modification and removal of entities

Modification of entities work in a similar way to what we already explained. The only difference is that modification events have two Ring entities (the subject of change and the result of the change) instead of only the subject. The process of dependency extraction is the same, but in this case is the code of the result of the change that will be parsed.

For deletions, the only events we consider as dependencies are deletions of entities held by a deleted holder. For example, if a method m from class C was deleted and then class C was also deleted. The deletion of m will be added to the candidate set of the deletion of C.

4. Implementation details

4.1 Anatomy of an Ombu entry

An entry in an Ombu file has a content, which can be any object, and a dictionary of tags. In the specific case of Epicea, the content is a change event [Figure 6]. The tag dictionary is used to store metadata like the author and the time of the change.

Another thing that is stored in the tag dictionary is the prior reference. Each entry has a reference to the prior change and this is the way the changes are linked as a list. We can use this mechanism to persist the dependency information for each entry.

As a second step, it is also desirable to be able to get the entries that contain the subject of change for each one of the dependencies.

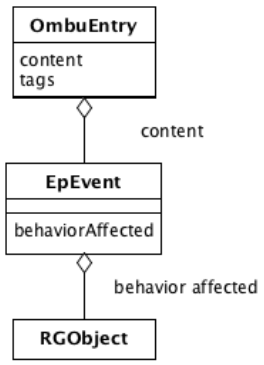


Figure 6. Anatomy of an Ombu entry

4.2 Retrieval of class holders

Epicea events can contain one or two RGObjects. Addition and removal of entries contain the new entity, while modifications contain the subject and the result. We defined the class holder in our model but it doesn't exist as a first-class object in the event. In these cases, the event only knows the name of the holding class. Therefore, we have to look in the log for the event of the creation of the holding class.

4.3 Retrieval of entries containing subjects of change

Dependencies are defined between affected entities: classes, methods and so on. Once the dependencies for an affected entity have been established, we have to find the events that affected those entities.

For example, let's suppose that we modified the definition of a class A. A new EpClassModification event object will be created and it will contain two instances of RGClassDefinition: one that represents the old class definition and another to represent the new one [Figure 1]. We have to find an entry that contains the event with the RGClassDefinition that represents the creation of the class A.

Almost all Epicea events are created with an RGObject as an internal collaborator [5]. Therefore, to have access to those entities, we can keep them in a multimap indexed by selector. The maintenance of the multimap (addition, changing and removal of entries) will be made at the moment of the event creation. And it won't be necessary to scan the complete log to find the related entries.

Since the events and entities are already present in the current Epicea implementation, the only additional objects that will be added are the dependencies.

5. Future work

In this section we describe some improvements to our initial solution.

5.1 Events vs. entries

In the current Epicea implementation, the Ombu entries are the ones that are linked through the prior reference. One of the limitations of this approach is that overlapping entries repeat the code through the related entries. For example, if we have an entry A with a class definition and an entry B with a modification to that class definition, B will contain all the code defined in A instead of having just a reference to it.

Another option would be to move the references to the event level. The model would be more sound from a semantic point of view and we can replace the duplicated code for a reference to the underlying event.

5.2 Visualization of dependencies

Once the extensions for the Epicea model are in place, we can modify the Log Browser to display the relationship between the entries in a graphical way.

One option is to draw lines between the entries in the Log Browser, as the GitK tool does. Another one, possibly more complex, is to add a tab in the lower panel that shows a dependency tree. This approach can be found in m2eclipse [6].

5.3 Performance test and optimization

It is desirable to test the performance in terms of execution time and memory consumption of this new features when they are used with a Log that contains several entries.

One alternative to reduce the memory footprint is to implement the multimap using a *Trie*. The keys will be the entity names, but all entities with a common prefix in their selectors will share that part of the key. The worst-case access time for a given selector would be the length of the longest selector defined in the system.

6. Conclusion

Detection of dependencies between changes modelled as first-class objects are a very important feature of code-history trackers, since it reduces the effort of the developer to perform tasks like atomic commits.

[5] The exception are expression evaluations, which are only strings evaluated by the compiler and don't have an associated Ring object.

[6] `https://www.eclipse.org/m2e/`

In this paper we defined a simple criteria to evaluate four different code-history trackers. We also present a solution to model dependencies between changes that makes Epicea feature-complete from the point of view of the aforementioned criteria.

References

[DAB+11] Stéphane Ducasse, Nicolas Anquetil, Usman Bhatti, Andre Cavalcante Hora, Jannik Laval, and Tudor Girba. MSE and FAMIX 3.0: an interexchange format and source code model family. Technical report, RMod – INRIA Lille-Nord Europe, 2011.

[DCD13] Martín Dias, Damien Cassou, and Stéphane Ducasse. Representing code history with development environment events. In *IWST'13: International Workshop on Smalltalk Technologies 2013*, 2013.

[EVC+07] Peter Ebraert, Jorge Vallejos, Pascal Costanza, Ellen Van Paesschen, and Theo D'Hondt. Change-oriented software engineering. In *Proceedings of the 2007 international conference on Dynamic languages: in conjunction with the 15th International Smalltalk Joint Conference*, ICDL '07, pages 3–24. ACM, 2007.

[Gol84] Adele Goldberg. *Smalltalk 80: the Interactive Programming Environment*. Addison Wesley, Reading, Mass., 1984.

[RL07] Romain Robbes and Michele Lanza. A change-based approach to software evolution. *Electronic Notes in Theoretical Computer Science*, 166:93–109, January 2007.

[STCH12] Bastian Steinert, Marcel Taeumel, Damien Cassou, and Robert Hirschfeld. Adopting design practices for programming. In *Design Thinking Research*. Springer, 2012.

[UG12] Verónica Uquillas Gómez. *Supporting Integration Activities in Object-Oriented Applications*. PhD thesis, Vrije Universiteit Brussel - Belgium & Université Lille 1 - France, October 2012.

[UGDD12] Verónica Uquillas Gómez, Stéphane Ducasse, and Theo D'Hondt. Ring: a unifying meta-model and infrastructure for Smalltalk source code analysis tools. *Journal of Computer Languages, Systems and Structures*, 38(1):44–60, April 2012.

Understanding Pharo's global state to move programs through time and space

Guillermo Polito, Noury Bouraqadi, Stéphane Ducasse, Luc Fabresse

Mines-Telecom Institute, Mines Douai
RMOD INRIA Lille Nord Europe

Abstract

Code mobility is a mechanism that allows the migration of running programs between different environments. Such migration includes amongst others the migration of application data and resources. Application's data is is usually composed by elements of different nature: from printers and files, to framework and domain objects. This application data will be transported along with the code of its program in space (when serialized and deployed in another environment) or time (when a new session is started in a different point of time). The main problem when moving around code resides, in our understanding, to *global* state. While unreferenced leaf objects are garbage collected, those referenced (transitively) by some global object will remain alive.

In order to support code mobility in time and space, we need to understand how global application data is used. With this purpose, we study and classify Pharo's global state. This classification uncovers some common patterns and provides a first insight on how global state should be managed, specially in code mobility scenarios. As a minor contribution, we also discuss solutions to each of the found categories.

1. Introduction

Code mobility is a mechanism that allows the migration of programs between different environments. It provides support for *e.g.,* load balancing, adjusting an application's resources dynamically and functionality customization. Fuggetta et al. define informally code mobility as the capability to rebind a piece of code with the location it is running [**?**]. Such rebinding may consist, depending on the style of mobility, in the mobility of execution state, application data and resources, or both of them. Execution state mobility is the ability to suspend the actual execution of a program and transfer its internal execution information (*e.g.,* code, execution stacks, instruction pointers) to some other environment. Data mobility is the ability to transfer the application's data (*e.g.,* objects, database connections, files) between different environments.

Application data is usually composed by elements of different nature. Files are used for configuration and logging. Network connections such as sockets are used to communicate with remote systems. External libraries provide with code reuse. We can also find objects local to the application, of two different categories: domain objects modeling the application's specific concerns and application objects modeling those concerns that are cross-cutting between applications.

In our experience manipulating the language kernel of Pharo, we identified several cases where data mobility presents some issues. We can generalize those issues as mobility either *in time* (*i.e.,* creating or recreating a program), or *in space* (*i.e.,* moving a program between different environments):

Transporting code in space. When moving a program from one environment to another one, some of its state becomes invalid. For example, files existing in one machine will not exist in some other. Because of this, the migration mechanism should be aware of the state it migrates, to either reinitialize it, re-bind it in the new environment, or by keep it with its same value [**?**].

Transporting code in time. Image-based systems allow one to persist the state of a program to restart it at some other point of time from the last check-point, introducing the idea of *program sessions*: every time the system is restarted, a new *session* is started. These programming sessions introduce the concern of *session specific state i.e.,* state that is only valid during a programming ses-

sion. The mechanism in charge of stopping and restarting the image has to recognize the session specific state to re-initialize or rebind it every time a new session is started.

Creating for the first time. The initial creation of the system is a combination of transporting the program in space and time, since the issues of both appear in it. When creating or recreating the language kernel from scratch, for example during a bootstrap process [**?**], we must deal with its initialization. All the initial objects must be created, and their state is initialized by either binding it for the first time to some resource or assigning it some value. This state should be initialized in a proper order.

One of the main problems when moving code around resides in the existence of *global* state. While unreferenced leaf objects are garbage collected, those referenced (transitively) by some global object will remain alive. Because of this, we focus our attention on global state (cf. Section 2). Migrating global state in the cases described above in a generic way shows itself challenging (cf. Section 3). Global state is used for many different and unrelated purposes in the Pharo base libraries *e.g.,* from caches to constants values. Also, the intention of such usage is not explicit in the source code: its identification requires the developer to read the complete implementation.

In this paper we present an empirical study on the global state of Pharo base libraries. Our contribution is twofold:

- We present a classification of the usage of global state, identifying patterns built with global state constructs in Pharo (cf. Section 4).
- We discuss our findings and solutions to the issues we found. Our main goal with this is to make explicit those patterns. In such a way, client libraries and frameworks in charge of program migration can be simplified. (cf. Section 6 and Section 5).

2. Background

Global state is a simple and handy mechanism to share state between different objects. It is also a simple persistency mechanism: state hold by it will persist as long as the program is alive and running. Additionally, in image-based systems as Pharo it will remain alive through different program executions because the image persists its state taking as root the global objects. In this paper we put focus on global state because of this persistency property.

Global state is indeed not a bad mechanism per se, and is often used in applications to implement globally needed concerns. For example, Pharo implements through it a global process scheduler and the system dictionary holding all classes. However, its usage is discouraged in general terms because it introduces hidden dependencies in the software it is used.

2.1 Global State in Pharo

Global state in Pharo can be expressed in many forms with many constructs of the language. In this paper we will focus on the elements we present following. Note that equivalent language constructs can be found in other languages such as Java (for example, with static variables).

Global Variables. Global variables are variables that share their values to all objects in the system. A global variable can be accessed from any method, from any object. In Pharo, these kind of variables are stored in the global SystemDictionary object. Global variables may reference either (a) global instances such as Processor or Smalltalk, either (b) Classes and Traits.

Class Variables. Class variables are variables that belong to a class. These variables can be accessed by both classes and instances from the hierarchy below its owner class. Their value is shared between all the objects that can access them. In Pharo, these kind of variables are stored in a Dictionary object in its owner class.

Class Instance Variables. Class instance variables are instance variables of the classes. Their value is not directly accessible from subclasses and subinstances of the class. However, they are often made globally accessible with accessors.

Shared Pools. Shared pools are sets of class variables shared amongst many classes. Their values are accessible to all classes (and their instances) that import the shared pool. In Pharo, Shared Pools as implemented as classes treated specially by the compiler at binding time.

Method Literals. Each method contains a collection of those literal objects used in in *e.g.,* strings, literal arrays or numbers. As classes are globally accessible, their methods are too, and so their literals.

2.2 About the State in Image-Based Systems

Pharo, as a Smalltalk inspired language, is an image-based language such as Lisp. Image-based languages present the following two main properties: direct object manipulation and persistence. Direct object manipulation provides with instant feedback during development and a flexible way to understand the state of applications. Persistence allows one to store those changes made by direct object manipulation without the need of recreating the system every time it is started. Indeed, an image-based language can be persisted and restarted later on, possibly in another machine. We refer as a *session* to the time elapsed between the startup of an image and its shutdown.

These programming sessions have *session specific state*, *i.e.,* state that is valid only within a session. For example, we can name as such file and socket descriptors, handles to external libraries, operating system information, and time and date information. These kind of objects become invalid

when their session is finished. Using them in an invalid state may lead to unexpected behavior, exceptions and virtual machine crashes. The language runtime must ensure that this state is correctly handled on session startup and shutdown: *e.g.,* reinitialize it or discard it.

3. Motivation

In this section we show why understanding and making explicit the usage of global state is important. We introduce first an example based on two Pharo's cache implementations, and their problems. Then, we explore three different situations in which those problems are made more evident.

3.1 Problems on Global State Usage: an Example

To exemplify the problems on global state usage, we present here two different global cache implementations we find in Pharo 3.0. First, in Figure 1, we present a simplified version of the AST cache. Second, in Figure 2, we find an extract of the HelpIcon class, with the code related to an icon cache. By looking at these two ad-hoc implementations of caches, we identify the following issues:

Incompleteness. Both cache implementations were written to solve only particular issues. The AST cache presents weak references as it inherits from the WeakIdentityKeyDictionary class and also presents methods to be flushed. The icon cache does not present code for any of those features. None of them cover some concerns a cache may want to address such as specifying a maximum amount of elements or a recycling strategy (LRU, FIFO, etc.).

Non-Explicitness. In order to identify the examples as caches we need to read their code: the names of the classes and variables gives us an idea of its responsibility as caches. The default method in the AST cache hints us about having found also a singleton. This problem uncovers the existence of **hidden information** in the system. One cannot query the system to, for example, obtain a list of the existing caches in order to flush them, or make a report on their memory usage.

3.2 Creating Programs from Scratch: Bootstrapping

While bootstrapping Pharo [**?**], we must initialize the image's global state. We observed the need for an order in this initialization, showing off a hidden coupling between code pieces. For example, some global tables must be initialized before initializing the classes state, which in turn must be initialized before the rest of the language kernel (*i.e.,* the startup and shutdown lists, the main processes, etc.).

Since the global state language constructs are used for different concerns **implicitly**, it is difficult to discern wether they are responsibility of the language kernel, of basic libraries such as Collections, or other not-basic ones such as Networking. This makes the bootstrap process difficult to maintain. A lot of ad-hoc code should be written to handle

```
1  WeakIdentityKeyDictionary subclass: #ASTCache
2    classVariableNames: 'Default'.
3
4  ASTCache>>at: aCompiledMethod
5    ^ self
6      at: aCompiledMethod
7      ifAbsentPut: [ self newASTFor: aCompiledMethod ].
8
9  ASTCache>>newASTFor: aMethod
10   "creation of the AST..."
11
12 ASTCache>>reset
13   self removeAll.
14
15 ASTCache class>>default
16   ^ Default ifNil: [ Default := self new ].
17
18 ASTCache class>>shutDown
19   self default reset.
```

Figure 1. Simplified code of Pharo's AST cache.

```
1  Object subclass: #HelpIcons
2    classVariableNames: 'Icons'.
3
4  HelpIcons>>icons
5    ^ Icons ifNil: [Icons := Dictionary new]
6
7  HelpIcons>>iconNamed: aSymbol
8    ^ self icons at: aSymbol ifAbsentPut: [self perform: aSymbol]
9
10 HelpIcons>>refreshIcon
11   ^ "creates a new icon object"
```

Figure 2. Code of Pharo's Help Icon class with an icon cache.

the dependencies between the global state in Pharo's language kernel.

3.3 Transporting Programs in Time: Session Awareness

Image-based systems introduce the concern of *session specific state*. State holding references to for example, files, caches, or platform specific information, may become invalid when a new session is started in a different moment. Pharo presents a startup and a shutdown mechanism to support this. The language runtime raises events on its startup and shutdown. Classes subscribed to such events are notified and will execute some code according to the event. The handler of these events is responsibility of the class developer. This mechanism hides information in two different levels:

Dependencies between classes. The subscribed classes receive the startup and shutdown events in an explicitly defined order. This order is present in a list which is defined by the developers. This list express a dependency between classes *e.g.,* some classes must receive the startup event before others to satisfy its invariants. However, this

list does not actually express the reason of this dependency *i.e.,* which is the state or invariants that should be guaranteed before each class receives the proper event.

Semantics of class state. The startup and shutdown event handlers, which are in charge of the clean-up and reinitialization of some the global state, are written in an imperative fashion. This imperative fashion hides the semantics and invariants of this state.

This hidden information makes difficult to change the startup and shutdown mechanism. Some questions appear when doing so: Can we remove some class from these lists? Can we alter the order without changing the behavior? When we register a new class, in which position should we put it?

3.4 Transporting Programs in Space: Serialization

Migrating objects, and specially code (classes and methods), from one image to another requires in general customizations for the global state it carries and references. References to external classes and global variables may not be serialized but just re-binded in the new environment. Class variables containing constant value objects may be transported with the program. Session specific state should be re-initialized, as program migration implies session change also.

A migration mechanism needs information about the semantics of the state in migration, so it knows wether it should reinitialize it, re-bind it, or keep it as it is. As this information is **not usually explicitly available** in the program under migration, the developer must add it in the form of extensions or descriptions, external to the program. For example, the serialization library Fuel [**?**] presents special clusters to handle and customize the serialization of global variables and class variables. The user must customize these clusters externally.

4. Classification of Pharo's Global State

4.1 Classification Methodology

Our universe of study is the latest release of Pharo, Pharo 3.0. We selected as individuals to study all those usages of global state language constructs as we presented it in Section 2 *i.e.,* class variables and class instance variables, shared pools and global variables. For simplicity, we excluded from our analysis the classes referenced by global variables. We also excluded method literals because analyzing them would mean to read every single method in the language kernel.

The global state in Pharo is present mostly in ad-hoc implementations, making difficult the usage of automated methods for its classification. Since the goal of this paper is not to obtain an automatic classification, we built our classification using purely empirical observation: reading the code. We took each of the selected individuals, read all the code related to it and made a qualitative evaluation of it. We put special emphasis on the side-effects on such individuals, which showed useful to recognize the individual's semantics in the program.

As a result, we distinguished some patterns of usage, which lead us to the categories in Section 4.2. Note that the individuals under study can fall into more than of these categories *e.g.,* a cache made globally as a singleton. Also, to avoid noise we excluded from the classification those individuals whose role in the source code was very specific, thus they did not conform a representative category.

4.2 Categories

Constants. Constants are values that are initialized once and never updated. Pharo has no construct to express constant values. Thus, they are expressed using the other available constructs. This means that the semantics of constants must be ensured explicitly in the code or they are not ensured at all.

Settings and Configurable Default Values. Settings and configurable default values provide a single point to configure and share values amongst several instances. They are publicly accessible so they can be modified and customized by developers. Pharo uses settings to store for example maximum size of UI widgets, code completion configurations and network configurations.

Singletons. Singletons are well known objects globally accessible in the system [**?** **?**]. They are used to provide a single access point to some shared state or behavior. Pharo presents several different singleton implementations: global leaf objects (not classes nor traits) such as *e.g.,* the Processor or the Transcript, leaf objects stored in class variables or class instance variables often accessible through the uniqueInstance message, and some classes which are indeed used as singletons.

Caches. A cache is a buffer that stores duplicated information to reduce the consumption of resources such as CPU or memory. Caches store usually up to a maximum amount of elements, discarding old ones following a given strategy *e.g.,* First In First Out (FIFO) and Least Recently Used (LRU). Pharo presents several caches which store for example images, fonts and package metadata.

Registries. A registry stores a list of possible service providers and resolves which one of them is the appropriate to handle a task. They are usually used as a factory, to decouple the users of a service from a particular implementation. For example, a compiler registry may store all the compiler implementations available and provide a default one. A registry allows users to subscribe and unsubscribe services into it. For example, when a notification has to be shown to the end user, the UIManager registry decides how to show it according to its registered providers: either by using the standard output or the graphical user interface. Pharo uses registries to manage different kind of concerns such as the compiler suite, the fonts or the UI interactions.

Session Specific State. Session specific state is the global state that is tied to a particular session *e.g.,* information gathered from the current platform, file handles and library handles. This state should be reinitialized or reset when a new session is started either in a new machine or a different one, to avoid misbehaviors and unexpected errors.

Process Controllers. Process controllers manage the life cycle of well known processes such as the idle process, the user interface (UI) process or the low space watcher process. They control how and when these well known processes are started, terminated, suspended and resumed.

Finalizables. Resources external to the language, such as files, sockets, or handles to external libraries, must be finalized accordingly when they are garbage collected or new session are started. For such a task, the classes of those objects implement a finalization mechanism to be aware of garbage collections and handle such situations.

Graphical Resources. Graphical resources are objects such as images, fonts, icons or bitmaps. These resources are embedded in the system using the global state constructs. As such, there is no general solution to discard them or reload them.

4.3 Results and Discussion of Impact

Table 1 lists the results of applying each of our categories to our set of individuals under study: how many of them apply to each category. The details of such a classification can be found in the Appendix A.

These results present some particularities we should take into account before doing a deep analysis. First, the number of detected graphical resources does not really represent the reality. A lot of graphical resources are represented as byte arrays in method literals (which we did not measure because of its complexity). With respect to the numbers in our results, we can argue that they give us an idea of the impact produced by each category *i.e.,* code-migration libraries have to potentially handle each appearance of these patterns in an ad-hoc fashion, since they are not explicit in the source code. For example, if we would decide that on serialization all caches should be flushed, we must add custom code to handle each of the 43 caches.

Category	Amount satifying
Constants	1722
Settings and Configurable Default Values	236
Singletons	65
Graphical Resources*	47
Caches	43
Registries	31
Session Specific	27
Process Controllers	11
Finalizables	6

Table 1. Amount of individuals classified under each of the identified categories.

5. Discussion: a need for Reification

5.1 Concepts to Reify in Pharo.

Bouraqadi et al. [**?** **?**] presented already the need for the reification of resources used in a mobile code. The reification of resources provide support for an open architecture and facilitates the task of object migration. They also make explicit the concepts that are part of the program, providing with information the system can benefit from. We identify in particular the need for reification of the following elements part of our categories:

Processes. Pharo processes, although they are already objects, are managed from other objects. Process specific state is controlled by objects other than the process itself, breaking encapsulation. As such, the life-cycle of processes are tied to those objects that create them or keep their state. A first class representation of processes, on the other hand, will encapsulate the process specific state, avoid conflicts on its access, and provide a common interface for their manipulation.

Finalizables. First class finalizable resources provide a framework supporting uniform finalization and resource deallocation.

Caches. First class caches provide a uniform and complete implementation of caches libraries can rely upon. Additionally they will enable the system with introspection and self-modification of such caches.

Variables. First class variables, namely slots, were sketched by Verwaest et al. [**?**] and a first version introduced into Pharo 3.0. Slots introduce the ability to refine instance variables, give them specific behavior and annotate them with meta-information. Specialized slots can be used to implement *e.g.,* session specific state, constant values or settings.

5.2 Using explicit metaclasses.

Finding all singletons installed in the system could be easily achieved through the usage of explicit metaclasses [**?**] or traits [**?**]. Explicit metaclasses and traits allow the sharing of behavior between classes, and thus, they eliminate the need for ad-hoc implementations of *e.g.,* singletons. Additionally, reifying the singleton abstraction in the language, provides with the ability to query and act upon the installed singletons. Implementing them with traits, however, presents as main limitation that the current trait implementation in Pharo is stateless. Thus, it does not allow to express class variables to hold the singleton instance.

6. Discussion: Moving responsibilities to the language runtime

Within our classification, we understand there are some concerns that should be moved under the umbrella of the language or its runtime system. The language may provide its own abstractions for recurrent problems such as caching or registering services. This will provide with the proper and needed meta-information to handle services. Additionally, providing end users with correct and complete implementations will avoid the ad-hoc implementations with repeated logic.

6.1 Resource manager

As we noted in the results, graphical resources such as images, icons and fonts are present as globally accessible resources in Pharo. We can add also that Pharo's memory is occupied in great percentage by instantiated bitmaps[1] [**?**]. There is not, however, a possibility to inspect all available resources, understand their origin (the package, class and method that defines them), or recreate them from files. This poses the need for a resource manager.

A sketch implementation of such a resource manager was implemented as a in-memory file system. In such a prototype, each Pharo package contains an associated file system that stores resources of that package. Images, icons, configuration files, and other files are stored in this file system. Package resources can be accessed from within and outside the package in an structured way, and serialized along with its package.

6.2 Session manager

How session specific state is handled nowadays denotes the need for a session manager. Currently, in the presence of session specific state, the class that stores it has to be subscribed to the startup and shutdown events of the runtime system. These two events are used to reset and initialize the class state when a new session is started.

We sketched a session manager to ease the management of session specific state. First class instance variables (Slots) describe declaratively their initialization when a new session is started. Then, during the startup of a new session, the session manager will reinitialize each of these slots using their description. This session manager encapsulates the need for the startup and shutdown lists, and removes such responsibility from the developer.

7. Related Work

Fuggetta et al. [**?**] present also a classification of the state of mobile systems, but using as criteria the strategy used for migration. As such, their classification is orthogonal and complementary to ours. They present two properties to characterize the data to migrate

Transferrable. A transferrable element is the one that can be physically migrated *e.g.,* a file. Oppositely, a non transferrable one is the one that cannot be migrated, *e.g.,* as a printer.

Desirability to transfer it. An application can mark some data as *fixed* or *free* according to its needs. Fixed data is associated permanently with its original environment, while free data migration is allowed.

and three ways to bind an application to a given resource

By identifier. Resources binded by identifier are tied with a particular instance of a resource *e.g.,* a socket. When a program is migrated, all its resources binder by identifier are kept in their original environment. A network communication is enforced between them.

By value. Resources binded by value are interested in the value of a resource and not in their identity, *e.g.,* the contents of a file. These kind of resources can be copied along with the program upon migration.

By type. Resources binded by type are intended to provide some kind of service despite their value or identity *e.g.,* a display. These kind of resources are rebinded to local resources of the same type after migration.

Ungar et al. implemented a transporter for the Self programming language [**?**]. This transporter had to deal with many of the difficulties we presented above, in particular the lack of explicit usage information. They provided a generic solution to the problem: let the developer annotate the objects' slots to guarantee the desired state of the program upon a migration. However, a question remained: How should developers annotate the slots? To answer this question, they provided with a series of properties that must help in such analysis.

Does identity matter? The developer has to identify those objects whose identity matters, and those whose it doesn't. When identity matters, the transporter must ensure that references to the same object are kept the same after migration. When it does not, the transporter can simply duplicate the object.

An initial value must always be enforced? Some objects must be reinitialized every time they are migrated. This is for example the case of caches.

An object must be written in an abstract or concrete way? Some objects can be rebuilt as the result of an expression, while some others must be built by concretely enumerating its slots.

8. Conclusion and Future Work

In this paper we studied the usage of global state in Pharo. The study of global state is interesting since references kept from global state are persisted in image-based systems. Global state is also a concern in when working in

[1] 24.50% according to our measures in latest Pharo version

code mobility because resources globally available must be reinitialized or rebinded when code is migrated.

We present a classification of Pharo's global state based on its usage, and found many patterns that are recurrent in the kernel of the language, though not explicit in the code.We discuss how to make explicit these patterns so the language kernel can benefit from it, either by reifying them or moving some responsibilities to the language kernel.

This work is a first step to prepare Pharo to the mobile code world. To be able to transport Pharo programs either in time or space, the abstractions we found should be made explicit in the language, and so, libraries and frameworks can take advantage of them. As future work we also consider that the discussed sketches have to be iterated and developed further.

Acknowledgements

This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council, FEDER through the 'Contrat de Projets Etat Region (CPER) 2007-2013', the Cutter ANR project, ANR-10-BLAN-0219 and the MEALS Marie Curie Actions program FP7-PEOPLE-2011- IRSES MEALS.

A. Appendix: Classification

A.1 Finalizables

FileHandle -> #Registry
FT2Handle -> #Registry
Socket -> #Registry
StandardFileStream -> #Registry
WeakRegistry -> #Default

A.2 Process Controllers

CPUWatcher -> #CurrentCPUWatcher
Delay -> #TimerEventLoop
MessageTally -> #Timer
MorphicUIManager -> #UIProcess
ProcessBrowser -> #SuspendedProcesses
ProcessBrowser -> #WellKnownProcesses
ProcessorScheduler -> #BackgroundProcess
SmalltalkImage -> #LowSpaceProcess
UpdateStreamer -> #UpdateDownloader
WeakArray -> #FinalizationProcess

A.3 Registries

Beeper -> #default
ChangeSet -> #AllChangeSets
ChangeSet -> #current
EncodedCharSet -> #EncodedCharSets
ExternalDropHandler -> #DefaultHAndler
ExternalDropHandler -> #RegisteredHandlers
FileServices -> #FileReaderRegistry
FreeTypeFontProvider -> #current
FreeTypeGlyphRenderer -> #current
HelpBrowser -> #DefaultHelpBrowser
LanguageEnvironment -> #ClipboardInterpreterClass
LanguageEnvironment -> #Current
LanguageEnvironment -> #FileNameConverter
LanguageEnvironment -> #InputInterpreterClass
LanguageEnvironment -> #KnownEnvironments
LanguageEnvironment -> #SystemConverter
Locale -> #KnownLocales
MCPackageManager -> #registry
MCServerRegistry -> #registry
MetacelloProjectRegistration -> #registry
Nautilus -> #PluginClasses
PluggableTextMorph -> #StylingClass
RBProgramNode -> #FormatterClass
RGFactory -> #CurrentFactories
SmalltalkImage -> #CompilerClass
SmalltalkImage -> #Tools
SoundSystem -> #Current
TestResource -> #current
UIManager -> #Default
UITheme -> #Current
ZnServer -> #ManagedServers
ZnSingleThreadedServer -> #Default

A.4 Caches

ASTCache -> #default
AbstractMethodWidget -> #MethodsIconsCache
AbstractNautilusUI -> #ClasssesIconsCache
AbstractNautilusUI -> #GroupsIconsCache
AbstractNautilusUI -> #PackagesIconsCache
BitBlt -> #CachedFontColorMaps
BitBlt -> #ColorConvertingMaps
CairoBackendCache -> #soleInstance
Color -> #CachedColormaps
Color -> #MaskingMap
FreeTypeCache -> #current
GLMUIThemeExtraIcons -> #icons
GradientFillStyle -> #PixelRampCache
HelpIcons -> #Icons
KomitClass -> #classes
KomitMethod -> #methods
KomitPackage -> #packages
KomitRemote -> #icon
Komitter -> #lastMessage
LoadingMorphState -> #image
LogicalFont -> #all
MCDefinition -> #Instances
MCGitHubRepository -> #DownloadCache
MCMethodDefinition -> #Definitions
MCSaveVersionDialog -> #PreviousMessages
NECSymbols -> #cachedSymbols
RPackageSet -> #cachePackages
ScrollBar -> #ArrowImagesCache
ScrollBar -> #BoxesImagesCache
SettingDeclaration -> #ValueListCache
SingleCodeCriticResultList -> #icons
SugsSuggestionFactory -> #collectorForAll
SugsSuggestionFactory -> #collectorForAssignment
SugsSuggestionFactory -> #collectorForClassVariable
SugsSuggestionFactory -> #collectorForClass
SugsSuggestionFactory -> #collectorForInstancesVariable
SugsSuggestionFactory -> #collectorForLiteral
SugsSuggestionFactory -> #collectorForMessage
SugsSuggestionFactory -> #collectorForMethod
SugsSuggestionFactory -> #collectorForSourceCode
SugsSuggestionFactory -> #collectorForTemporaryVariable
SugsSuggestionFactory -> #collectorForUndeclaredVariable

A.5 Graphical Resources

AbstractMethodWidget -> #MethodsIconsCache
AbstractNautilusUI -> #ClasssesIconsCache
AbstractNautilusUI -> #GroupsIconsCache
AbstractNautilusUI -> #PackagesIconsCache
Cursor -> #BlankCursor
Cursor -> #BottomLeftCursor
Cursor -> #BottomRightCursor
Cursor -> #CornerCursor
Cursor -> #CrossHairCursor
Cursor -> #CurrentCursor
Cursor -> #DownCursor
Cursor -> #MarkerCursor
Cursor -> #MenuCursor
Cursor -> #MoveCursor
Cursor -> #NormalCursor

Cursor -> #OriginCursor
Cursor -> #OverEditableText
Cursor -> #ReadCursor
Cursor -> #ResizeLeftCursor
Cursor -> #ResizeTopCursor
Cursor -> #ResizeTopLeftCursor
Cursor -> #ResizeTopRightCursor
Cursor -> #RightArrowCursor
Cursor -> #SquareCursor
Cursor -> #TargetCursor
Cursor -> #TopLeftCursor
Cursor -> #TopRightCursor
Cursor -> #UpCursor
Cursor -> #WaitCursor
Cursor -> #WebLinkCursor
Cursor -> #WriteCursor
Cursor -> #XeqCursor
FreeTypeCache -> #current
FreeTypeSettings -> #current
GLMUIThemeExtraIcons -> #icons
HelpIcons -> #Icons
IconicButton -> #DefaultGraphics
ImageMorph -> #DefaultForm
LogicalFontManager -> #current
RemotesManager -> #addRemoteIcon
RemotesManager -> #editRemoteIcon
RemotesManager -> #removeRemoteIcon
ScrollBar -> #ArrowImagesCache
ScrollBar -> #BoxesImagesCache
SingleCodeCriticResultList -> #icons
Transcripter -> #Icon
TransferMorph -> #CopyPlusIcon

A.6 Session Specific State

MCGitHubRepository -> #DownloadCache
MCCacheRepository -> #default
DiskStore -> #CurrentFS
NOCCompletionTable -> #table
NOCCompletionTable -> #classTable
Locale -> #Current
Locale -> #CurrentPlatform
DateAndTime -> #LocalTimeZone
FT2Handle -> #Session
FileLocator -> #Resolver
FileStream -> #Stdin
FileStream -> #Stdout
FileStream -> #TheStdioHandles
FileStream -> #StdioFiles
FileStream -> #Stderr
LanguageEnvironment -> #SystemConverter
LanguageEnvironment -> #FileNameConverter
UUIDGenerator -> #Default
VirtualMachine -> #WordSize
WeakFinalizationList -> #HasNewFinalization
AthensCairoSurface -> #uniqueSession
AthensCairoSurface -> #dispatch
AthensCairoSurface -> #dispatchStruct
CairoLibraryLoader -> #session
CairoLibraryLoader -> #libHandle
Session -> #current
MultiByteFileStream -> #LineEndDefault

A.7 Singletons

ASTCache -> #default
ActiveEvent -> #ActiveEvent
ActiveHand -> #ActiveHand
ActiveWorld -> #ActiveWorld
Author -> #uniqueInstance
BorderStyle -> #Default
CPUWatcher -> #CurrentCPUWatcher
CairoBackendCache -> #soleInstance
ChangesLog -> #DefaultInstance
Clipboard -> #Default
CommandLineArguments -> #singleton
CriticWorkingConfiguration -> #Current
Display -> #Display
EditorFindReplaceDialogWindow -> #Singleton
EmptyLayout -> #instance
FreeTypeCache -> #current
FreeTypeSettings -> #current
IdentityTransform -> #Default
InputEventFetcher -> #Default
KMBuffer -> #uniqueInstance
KMPragmaKeymapBuilder -> #UniqueInstance
KMRepository -> #Singleton
KomitterManager -> #instance
LayoutEmptyScope -> #instance
LogicalFontManager -> #current
MBConfigurationRoot -> #Current
MCFileTreeFileUtils -> #Current
MCRepositoryGroup -> #default
MCServerRegistry -> #uniqueInstance
MetacelloPlatform -> #Current
NBExternalResourceManager -> #soleInstance
NECController -> #uniqueInstance
NNavNavigation -> #Instance
NNavNavigation -> #Instance
NativeBoost -> #Current
NautilusMonticello -> #Default
OSPlatform -> #Current
PackageOrganizer -> #default
PharoFilesOpener -> #Default
PharoTutorial -> #Instance
ProcessSpecificVariable -> #soleInstance
Processor -> #Processor
RBRefactoringManager -> #Instance
RBRefactoryChangeManager -> #Instance
RPackageOrganizer -> #default
RecentMessageList -> #UniqueInstance
Sensor -> #Sensor
SharedValueHolder -> #instance
Smalltalk -> #Smalltalk
SoundTheme -> #Current
SourceFiles -> #SourceFiles
Spotlight -> #Current
StartupPreferencesLoader -> #UniqueInstance
SystemAnnouncer -> #announcer

SystemOrganization -> #SystemOrganization
SystemProgressMorph -> #UniqueInstance
SystemVersion -> #Current
Transcript -> #Transcript
UUIDGenerator -> #Default
Undeclared -> #Undeclared
UserManager -> #default
VTermOutputDriver -> #stderrTerminalInstance
VTermOutputDriver -> #stdoutTerminalInstance
World -> #World
ZnNetworkingUtils -> #Default

A.8 Settings and Configurable Default Values

AbstractNautilusUI -> #NextFocusKey
AbstractNautilusUI -> #PreviousFocusKey
AlphaImageMorph -> #DefaultImage
BalloonMorph -> #BalloonFont
CCompilationContext -> #WarningAllowed
CPUWatcher -> #CpuWatcherEnabled
ChangeSet -> #DefaultChangeSetDirectoryName
ChangeSet -> #MustCheckForSlips
CodeHolder -> #AnnotationRequests
CodeHolder -> #BrowseWithPrettyPrint
CodeHolder -> #DecorateBrowserButtons
CodeHolder -> #DiffsInChangeList
CodeHolder -> #DiffsWithPrettyPrint
CodeHolder -> #OptionalButtons
CodeHolder -> #ShowAnnotationPane
CodeHolder -> #SmartUpdating
CommandLineUIManager -> #SnapshotErrorImage
DangerousClassNotifier -> #enabled
Deprecation -> #RaiseWarning
Deprecation -> #ShowWarning
DialogItemsChooserUI -> #alreadySearchedSelectedItemsListMaxSize
DialogItemsChooserUI -> #alreadySearchedUnselectedItemsListMaxSize
DisplayScreen -> #DeferringUpdates
DisplayScreen -> #DisplayChangeSignature
DisplayScreen -> #LastScreenModeSelected
DisplayScreen -> #ScreenSave
Editor -> #BlinkingCursor
Editor -> #CmdKeysInText
Editor -> #DumbbellCursor
Editor -> #SkipOverMultipleSpaces
EyeInspector -> #useAutoRefresh
FLCompiledMethodCluster -> #transformationForSerializing
FinderUI -> #Icon
FinderUI -> #searchedTextListMaxSize
Form -> #FloodFillTolerance
FreeTypeSettings -> #UpdateFontsAtImageStartup
FreeTypeSystemSettings -> #LoadFT2Library
GrowlMorph -> #DefaultBackgroundColor
GrowlMorph -> #Position
HaloMorph -> #CurrentHaloSpecifications
HaloMorph -> #HaloEnclosesFullBounds
HaloMorph -> #HaloWithDebugHandle
HaloMorph -> #ShowBoundsInHalo
HandMorph -> #DoubleClickTime
HandMorph -> #NormalCursor
HandMorph -> #ShowEvents
HandMorph -> #UpperHandLimit
Heap -> #sortBlock
LongTestCase -> #RunLongTestCases
MBInfo -> #ValidateAll
MCDirectoryRepository -> #DefaultDirectoryName
MCFileRepositoryInspector -> #Order
MCFileTreeRepository -> #defaultPackageExtension
MCFileTreeRepository -> #defaultPropertyFileExtension
MCGitHubRepository -> #CacheDirectory
MCMethodDefinition -> #InitializerEnabled
MCWorkingCopyBrowser -> #Order
MCWorkingCopyBrowser -> #ShowOnlyRepositoriesFromWorkingCopy
MCWorkingCopyBrowser -> #repositorySearchMaxSize
MCWorkingCopyBrowser -> #workingCopySearchMaxSize
MessageDialogWindow -> #AutoAccept
MessageTally -> #DefaultPollPeriod
MetacelloCommonMCSpecLoader -> #RetryPackageResolution
MetacelloScriptEngine -> #DefaultRepositoryDescription
MetacelloScriptEngine -> #DefaultVersionString
MonticelloRepositoryBrowser -> #Order
Morph -> #CmdGesturesEnabled
Morph -> #CycleHalosBothDirections
Morph -> #DefaultYellowButtonMenuEnabled
Morph -> #HalosEnabled
MorphicModel -> #KeyboadFocusOnMouseDown
MorphicModel -> #MouseOverForKeyboardFocus
NECPreferences -> #backgroundColor
NECPreferences -> #captureNavigationKeys
NECPreferences -> #caseSensitive
NECPreferences -> #enabled
NECPreferences -> #expandPrefixes
NECPreferences -> #popupAutomaticDelay
NECPreferences -> #popupShowAutomatic
NECPreferences -> #popupShowWithShortcut
NECPreferences -> #smartCharactersMapping
NECPreferences -> #smartCharactersWithDoubleSpace
NECPreferences -> #smartCharactersWithSingleSpace
NECPreferences -> #smartCharacters
NECPreferences -> #spaceAfterCompletion
NECPreferences -> #useEnterToAccept
NNavNavigation -> #UseArrowShortcuts
Nautilus -> #CommentPosition
Nautilus -> #HistoryMaxSize
Nautilus -> #OpenOnGroups
Nautilus -> #ShowAnnotationPane
Nautilus -> #ShowHierarchy
Nautilus -> #SwitchClassesAndPackages
Nautilus -> #WarningLimit
Nautilus -> #emptyCommentWarning
Nautilus -> #maxSize
Nautilus -> #populateMethodList
Nautilus -> #useOldStyleKeys
NautilusRefactoring -> #PromptOnRefactoring
NetNameResolver -> #DefaultHostName
NetworkSystemSettings -> #BlabEmail
NetworkSystemSettings -> #HTTPProxyExceptions

NetworkSystemSettings -> #HTTPProxyPort
NetworkSystemSettings -> #HTTPProxyServer
NetworkSystemSettings -> #ProxyPassword
NetworkSystemSettings -> #ProxyUser
NetworkSystemSettings -> #UseHTTPProxy
NetworkSystemSettings -> #UseNetworkAuthentication
ObjectExplorer -> #ShowIcons
PSMCPatchMorph -> #UsedByDefault
PackageTreeNautilus -> #ShowGroupsOnTop
Paragraph -> #InsertionPointColor
Path -> #absoluteWindowsPathRegex
PluggableButtonMorph -> #UseGradientLook
PluggableTextMorph -> #ShowTextEditingState
PluggableTextMorph -> #StylingClass
PluggableTextMorphWithLimits -> #DefaultWarningLimit
PolygonMorph -> #CurvierByDefault
PolymorphSystemSettings -> #DesktopColor
PolymorphSystemSettings -> #DesktopColor
PolymorphSystemSettings -> #DesktopGradientDirection
PolymorphSystemSettings -> #DesktopGradientDirection
PolymorphSystemSettings -> #DesktopGradientFillColor
PolymorphSystemSettings -> #DesktopGradientFillColor
PolymorphSystemSettings -> #DesktopGradientOrigin
PolymorphSystemSettings -> #DesktopGradientOrigin
PolymorphSystemSettings -> #DesktopImageFileName
PolymorphSystemSettings -> #DesktopImageFileName
PolymorphSystemSettings -> #DesktopLogoFileName
PolymorphSystemSettings -> #DesktopLogoFileName
PolymorphSystemSettings -> #DesktopLogo
PolymorphSystemSettings -> #DesktopLogo
PolymorphSystemSettings -> #ShowDesktopLogo
PolymorphSystemSettings -> #ShowDesktopLogo
PolymorphSystemSettings -> #UseDesktopGradientFill
PolymorphSystemSettings -> #UseDesktopGradientFill
PolymorphSystemSettings -> #usePolymorphDiffMorph
PolymorphSystemSettings -> #usePolymorphDiffMorph
ProgressBarMorph -> #DefaultHeight
ProgressBarMorph -> #DefaultWidth
ProportionalSplitterMorph -> #ShowHandles
RBConfigurableFormatter -> #CascadedMessageInsideParentheses
RBConfigurableFormatter -> #FormatCommentWithStatements
RBConfigurableFormatter -> #IndentString
RBConfigurableFormatter -> #IndentsForKeywords
RBConfigurableFormatter -> #KeepBlockInMessage
RBConfigurableFormatter -> #LineUpBlockBrackets
RBConfigurableFormatter -> #MaxLineLength
RBConfigurableFormatter -> #MethodSignatureOnMultipleLines
RBConfigurableFormatter -> #MinimumNewLinesBetweenStatements
RBConfigurableFormatter -> #MultiLineMessages
RBConfigurableFormatter -> #NewLineAfterCascade
RBConfigurableFormatter -> #NewLineBeforeFirstCascade
RBConfigurableFormatter -> #NewLineBeforeFirstKeyword
RBConfigurableFormatter -> #NewLinesAfterMethodComment
RBConfigurableFormatter -> #NewLinesAfterMethodPattern
RBConfigurableFormatter -> #NewLinesAfterTemporaries
RBConfigurableFormatter -> #NumberOfArgumentsForMultiLine
RBConfigurableFormatter -> #OneLineMessages
RBConfigurableFormatter -> #PeriodsAtEndOfBlock
RBConfigurableFormatter -> #PeriodsAtEndOfMethod
RBConfigurableFormatter -> #RetainBlankLinesBetweenStatements
RBConfigurableFormatter -> #StringFollowingReturn
RBConfigurableFormatter -> #StringInsideBlocks
RBConfigurableFormatter -> #StringInsideParentheses
RBConfigurableFormatter -> #TraditionalBinaryPrecedence
RBConfigurableFormatter -> #UseTraditionalBinaryPrecedenceForParentheses
RBRefactoring -> #RefactoringOptions
RBRefactoryChangeManager -> #UndoSize
RealEstateAgent -> #StaggerOffset
RealEstateAgent -> #StandardSize
RealEstateAgent -> #UsedStrategy
RecentMessageList -> #settingDropList
SHPreferences -> #CustomStyleTable
SHPreferences -> #Groups
SHPreferences -> #enabled
SHTextStylerST80 -> #styleTable
SHTextStylerST80 -> #textAttributesByPixelHeight
ScriptLoader -> #CheckImageSyncWithUpdate
SettingBrowser -> #regexpSearch
SettingBrowser -> #searchedTextList
SimpleEditor -> #CmdActions
SimpleEditor -> #ShiftCmdActions
SmalltalkImage -> #ShouldDownloadSourcesFile
SoundSystem -> #SoundEnabled
SoundSystem -> #SoundQuickStart
SoundTheme -> #UseThemeSounds
SpecDebugger -> #AlwaysOpenFullDebugger
SpecDebugger -> #ErrorRecursion
SpecDebugger -> #FilterCommonMessageSends
SpecDebugger -> #LogDebuggerStackToFile
SpecDebugger -> #LogFileName
SpecDebuggerStack -> #DoItFilterEnabled
SpecDebuggerStack -> #FilterDictionary
SpecDebuggerStack -> #KCFilterEnabled
SpecDebuggerStack -> #NilSelectorsFilterEnabled
StandardFonts -> #ButtonFont
StandardFonts -> #CodeFont
StandardFonts -> #HaloFont
StandardFonts -> #ListFont
StandardFonts -> #MenuFont
StandardFonts -> #WindowTitleFont
StartupPreferencesLoader -> #AllowStartupScript
StringMorph -> #EditableStringMorph
SystemProgressMorph -> #horizontalPosition
SystemProgressMorph -> #verticalPosition
SystemWindow -> #CloseBoxImage
SystemWindow -> #CollapseBoxImage
SystemWindow -> #FullscreenMargin
TaskListMorph -> #KeepOpen
TaskbarMorph -> #ShowTaskbar
TaskbarMorph -> #ShowWindowPreview
TextDiffBuilder -> #IgnoreLineEndings
TextDiffBuilder -> #InsertTextAttributes
TextDiffBuilder -> #NormalTextAttributes
TextDiffBuilder -> #RemoveTextAttributes
TextEditor -> #CaseSensitiveFinds

TextEditor -> #UseFindReplaceSelection
TextEditor -> #UseSecondarySelection
TextEditor -> #UseSelectionBar
TextEditor -> #cmdActions
TextEditor -> #shiftCmdActions
TextEntryDialogWindow -> #MinimumWidth
UITheme -> #defaultSettings
UserInterruptHandler -> #CmdDotEnabled
Week -> #StartDay
WorldState -> #CanSurrenderToOS
WorldState -> #DebugShowDamage
WorldState -> #DesktopMenuPragmaKeyword
WorldState -> #DesktopMenuTitle
WorldState -> #EasySelectingWorld
WorldState -> #MinCycleLapse
WorldState -> #ServerMode
WorldState -> #ShowUpdateOptionInWorldMenu
ZnConstants -> #DefaultMaximumEntitySize
ZnServer -> #AlwaysRestart

A.9 Constants

AJConstants -> #CcA
AJConstants -> #CcABOVE
AJConstants -> #CcABOVEEQUAL
AJConstants -> #CcAE
AJConstants -> #CcB
AJConstants -> #CcBE
AJConstants -> #CcBELOW
AJConstants -> #CcBELOWEQUAL
AJConstants -> #CcC
AJConstants -> #CcE
AJConstants -> #CcEQUAL
AJConstants -> #CcFPNOTUNORDERED
AJConstants -> #CcFPUNORDERED
AJConstants -> #CcG
AJConstants -> #CcGE
AJConstants -> #CcGREATER
AJConstants -> #CcGREATEREQUAL
AJConstants -> #CcL
AJConstants -> #CcLE
AJConstants -> #CcLESS
AJConstants -> #CcLESSEQUAL
AJConstants -> #CcNA
AJConstants -> #CcNAE
AJConstants -> #CcNB
AJConstants -> #CcNBE
AJConstants -> #CcNC
AJConstants -> #CcNE
AJConstants -> #CcNEGATIVE
AJConstants -> #CcNG
AJConstants -> #CcNGE
AJConstants -> #CcNL
AJConstants -> #CcNLE
AJConstants -> #CcNO
AJConstants -> #CcNOCONDITION
AJConstants -> #CcNOOVERFLOW
AJConstants -> #CcNOTEQUAL
AJConstants -> #CcNOTSIGN
AJConstants -> #CcNOTZERO
AJConstants -> #CcNP
AJConstants -> #CcNS
AJConstants -> #CcNZ
AJConstants -> #CcO
AJConstants -> #CcOVERFLOW
AJConstants -> #CcP
AJConstants -> #CcPARITYEVEN
AJConstants -> #CcPARITYODD
AJConstants -> #CcPE
AJConstants -> #CcPO
AJConstants -> #CcPOSITIVE
AJConstants -> #CcS
AJConstants -> #CcSIGN
AJConstants -> #CcZ
AJConstants -> #CcZERO
AJConstants -> #InstCMOVA
AJConstants -> #InstJA
AJConstants -> #O64Only
AJConstants -> #OFM1
AJConstants -> #OFM10
AJConstants -> #OFM2
AJConstants -> #OFM24
AJConstants -> #OFM248
AJConstants -> #OFM4
AJConstants -> #OFM48
AJConstants -> #OFM4810
AJConstants -> #OFM8
AJConstants -> #OG16
AJConstants -> #OG163264
AJConstants -> #OG32
AJConstants -> #OG3264
AJConstants -> #OG64
AJConstants -> #OG8
AJConstants -> #OG8163264
AJConstants -> #OIMM
AJConstants -> #OMEM
AJConstants -> #OMM
AJConstants -> #OMMMEM
AJConstants -> #OMMXMM
AJConstants -> #OMMXMMMEM
AJConstants -> #ONOREX
AJConstants -> #OXMM
AJConstants -> #OXMMMEM
AJConstants -> #OpImm
AJConstants -> #OpLabel
AJConstants -> #OpMem
AJConstants -> #OpNONE
AJConstants -> #OpREG
AJConstants -> #PrefetchNTA
AJConstants -> #PrefetchT0
AJConstants -> #PrefetchT1
AJConstants -> #PrefetchT2
AJConstants -> #RIDEAX
AJConstants -> #RIDEBP
AJConstants -> #RIDEBX
AJConstants -> #RIDECX
AJConstants -> #RIDEDI
AJConstants -> #RIDEDX
AJConstants -> #RIDESI

AJConstants -> #RIDESP
AJConstants -> #RegCodeMask
AJConstants -> #RegGPB
AJConstants -> #RegGPD
AJConstants -> #RegGPQ
AJConstants -> #RegGPW
AJConstants -> #RegHighByteMask
AJConstants -> #RegMM
AJConstants -> #RegProhibitsRexMask
AJConstants -> #RegRequiresRexMask
AJConstants -> #RegTypeMask
AJConstants -> #RegX87
AJConstants -> #RegXMM
AJConstants -> #SegmentCS
AJConstants -> #SegmentDS
AJConstants -> #SegmentES
AJConstants -> #SegmentFS
AJConstants -> #SegmentGS
AJConstants -> #SegmentNONE
AJConstants -> #SegmentSS
AJConstants -> #SizeByte
AJConstants -> #SizeDQWord
AJConstants -> #SizeDWord
AJConstants -> #SizeQWord
AJConstants -> #SizeTWord
AJConstants -> #SizeWord
AJx86InstructionDescription -> #instructions
AJx86Registers -> #AH
AJx86Registers -> #AL
AJx86Registers -> #AX
AJx86Registers -> #BH
AJx86Registers -> #BL
AJx86Registers -> #BP
AJx86Registers -> #BPL
AJx86Registers -> #BX
AJx86Registers -> #CH
AJx86Registers -> #CL
AJx86Registers -> #CX
AJx86Registers -> #Codes
AJx86Registers -> #DH
AJx86Registers -> #DI
AJx86Registers -> #DIL
AJx86Registers -> #DL
AJx86Registers -> #DX
AJx86Registers -> #EAX
AJx86Registers -> #EBP
AJx86Registers -> #EBX
AJx86Registers -> #ECX
AJx86Registers -> #EDI
AJx86Registers -> #EDX
AJx86Registers -> #EIP
AJx86Registers -> #ESI
AJx86Registers -> #ESP
AJx86Registers -> #IP
AJx86Registers -> #MM0
AJx86Registers -> #MM1
AJx86Registers -> #MM2
AJx86Registers -> #MM3
AJx86Registers -> #MM4
AJx86Registers -> #MM5
AJx86Registers -> #MM6
AJx86Registers -> #MM7
AJx86Registers -> #R10
AJx86Registers -> #R10B
AJx86Registers -> #R10D
AJx86Registers -> #R10W
AJx86Registers -> #R11
AJx86Registers -> #R11B
AJx86Registers -> #R11D
AJx86Registers -> #R11W
AJx86Registers -> #R12
AJx86Registers -> #R12B
AJx86Registers -> #R12D
AJx86Registers -> #R12W
AJx86Registers -> #R13
AJx86Registers -> #R13B
AJx86Registers -> #R13D
AJx86Registers -> #R13W
AJx86Registers -> #R14
AJx86Registers -> #R14B
AJx86Registers -> #R14D
AJx86Registers -> #R14W
AJx86Registers -> #R15
AJx86Registers -> #R15B
AJx86Registers -> #R15D
AJx86Registers -> #R15W
AJx86Registers -> #R8
AJx86Registers -> #R8B
AJx86Registers -> #R8D
AJx86Registers -> #R8W
AJx86Registers -> #R9
AJx86Registers -> #R9B
AJx86Registers -> #R9D
AJx86Registers -> #R9W
AJx86Registers -> #RAX
AJx86Registers -> #RBP
AJx86Registers -> #RBX
AJx86Registers -> #RCX
AJx86Registers -> #RDI
AJx86Registers -> #RDX
AJx86Registers -> #RIP
AJx86Registers -> #RSI
AJx86Registers -> #RSP
AJx86Registers -> #SI
AJx86Registers -> #SIL
AJx86Registers -> #SP
AJx86Registers -> #SPL
AJx86Registers -> #ST0
AJx86Registers -> #ST1
AJx86Registers -> #ST2
AJx86Registers -> #ST3
AJx86Registers -> #ST4
AJx86Registers -> #ST5
AJx86Registers -> #ST6
AJx86Registers -> #ST7
AJx86Registers -> #XMM0
AJx86Registers -> #XMM1
AJx86Registers -> #XMM10
AJx86Registers -> #XMM11

AJx86Registers -> #XMM12
AJx86Registers -> #XMM13
AJx86Registers -> #XMM14
AJx86Registers -> #XMM15
AJx86Registers -> #XMM2
AJx86Registers -> #XMM3
AJx86Registers -> #XMM4
AJx86Registers -> #XMM5
AJx86Registers -> #XMM6
AJx86Registers -> #XMM7
AJx86Registers -> #XMM8
AJx86Registers -> #XMM9
AsyncFile -> #Busy
AsyncFile -> #ErrorCode
AthensBezierConverter -> #CollinearityEps
AthensBezierConverter -> #CurveAngleTolerance
AthensBezierConverter -> #DistanceEps
AthensCairoDefs -> #CAIRO_ANTIALIAS_BEST
AthensCairoDefs -> #CAIRO_ANTIALIAS_DEFAULT
AthensCairoDefs -> #CAIRO_ANTIALIAS_FAST
AthensCairoDefs -> #CAIRO_ANTIALIAS_GOOD
AthensCairoDefs -> #CAIRO_ANTIALIAS_GRAY
AthensCairoDefs -> #CAIRO_ANTIALIAS_NONE
AthensCairoDefs -> #CAIRO_ANTIALIAS_SUBPIXEL
AthensCairoDefs -> #CAIRO_EXTEND_NONE
AthensCairoDefs -> #CAIRO_EXTEND_PAD
AthensCairoDefs -> #CAIRO_EXTEND_REFLECT
AthensCairoDefs -> #CAIRO_EXTEND_REPEAT
AthensCairoDefs -> #CAIRO_FONT_SLANT_ITALIC
AthensCairoDefs -> #CAIRO_FONT_SLANT_NORMAL
AthensCairoDefs -> #CAIRO_FONT_SLANT_OBLIQUE
AthensCairoDefs -> #CAIRO_FONT_TYPE_FT
AthensCairoDefs -> #CAIRO_FONT_TYPE_QUARTZ
AthensCairoDefs -> #CAIRO_FONT_TYPE_TOY
AthensCairoDefs -> #CAIRO_FONT_TYPE_USER
AthensCairoDefs -> #CAIRO_FONT_TYPE_WIN32
AthensCairoDefs -> #CAIRO_FONT_WEIGHT_BOLD
AthensCairoDefs -> #CAIRO_FONT_WEIGHT_NORMAL
AthensCairoDefs -> #CAIRO_FORMAT_A1
AthensCairoDefs -> #CAIRO_FORMAT_A8
AthensCairoDefs -> #CAIRO_FORMAT_ARGB32
AthensCairoDefs -> #CAIRO_FORMAT_INVALID
AthensCairoDefs -> #CAIRO_FORMAT_RGB16_565
AthensCairoDefs -> #CAIRO_FORMAT_RGB24
AthensCairoDefs -> #CAIRO_HINT_METRICS_DEFAULT
AthensCairoDefs -> #CAIRO_HINT_METRICS_OFF
AthensCairoDefs -> #CAIRO_HINT_METRICS_ON
AthensCairoDefs -> #CAIRO_HINT_STYLE_DEFAULT
AthensCairoDefs -> #CAIRO_HINT_STYLE_FULL
AthensCairoDefs -> #CAIRO_HINT_STYLE_MEDIUM
AthensCairoDefs -> #CAIRO_HINT_STYLE_NONE
AthensCairoDefs -> #CAIRO_HINT_STYLE_SLIGHT
AthensCairoDefs -> #CAIRO_LINE_CAP_BUTT
AthensCairoDefs -> #CAIRO_LINE_CAP_ROUND
AthensCairoDefs -> #CAIRO_LINE_CAP_SQUARE
AthensCairoDefs -> #CAIRO_LINE_JOIN_BEVEL
AthensCairoDefs -> #CAIRO_LINE_JOIN_MITER
AthensCairoDefs -> #CAIRO_LINE_JOIN_ROUND
AthensCairoDefs -> #CAIRO_OPERATOR_ADD
AthensCairoDefs -> #CAIRO_OPERATOR_ATOP
AthensCairoDefs -> #CAIRO_OPERATOR_CLEAR
AthensCairoDefs -> #CAIRO_OPERATOR_COLOR_BURN
AthensCairoDefs -> #CAIRO_OPERATOR_COLOR_DODGE
AthensCairoDefs -> #CAIRO_OPERATOR_DARKEN
AthensCairoDefs -> #CAIRO_OPERATOR_DEST
AthensCairoDefs -> #CAIRO_OPERATOR_DEST_ATOP
AthensCairoDefs -> #CAIRO_OPERATOR_DEST_IN
AthensCairoDefs -> #CAIRO_OPERATOR_DEST_OUT
AthensCairoDefs -> #CAIRO_OPERATOR_DEST_OVER
AthensCairoDefs -> #CAIRO_OPERATOR_DIFFERENCE
AthensCairoDefs -> #CAIRO_OPERATOR_EXCLUSION
AthensCairoDefs -> #CAIRO_OPERATOR_HARD_LIGHT
AthensCairoDefs -> #CAIRO_OPERATOR_HSL_COLOR
AthensCairoDefs -> #CAIRO_OPERATOR_HSL_HUE
AthensCairoDefs -> #CAIRO_OPERATOR_HSL_LUMINOSITY
AthensCairoDefs -> #CAIRO_OPERATOR_HSL_SATURATION
AthensCairoDefs -> #CAIRO_OPERATOR_IN
AthensCairoDefs -> #CAIRO_OPERATOR_LIGHTEN
AthensCairoDefs -> #CAIRO_OPERATOR_MULTIPLY
AthensCairoDefs -> #CAIRO_OPERATOR_OUT
AthensCairoDefs -> #CAIRO_OPERATOR_OVER
AthensCairoDefs -> #CAIRO_OPERATOR_OVERLAY
AthensCairoDefs -> #CAIRO_OPERATOR_SATURATE
AthensCairoDefs -> #CAIRO_OPERATOR_SCREEN
AthensCairoDefs -> #CAIRO_OPERATOR_SOFT_LIGHT
AthensCairoDefs -> #CAIRO_OPERATOR_SOURCE
AthensCairoDefs -> #CAIRO_OPERATOR_XOR
AthensCairoDefs -> #CAIRO_STATUS_CLIP_NOT_REPRESENTABLE
AthensCairoDefs -> #CAIRO_STATUS_DEVICE_ERROR
AthensCairoDefs -> #CAIRO_STATUS_DEVICE_TYPE_MISMATCH
AthensCairoDefs -> #CAIRO_STATUS_FILE_NOT_FOUND
AthensCairoDefs -> #CAIRO_STATUS_FONT_TYPE_MISMATCH
AthensCairoDefs -> #CAIRO_STATUS_INVALID_CLUSTERS
AthensCairoDefs -> #CAIRO_STATUS_INVALID_CONTENT
AthensCairoDefs -> #CAIRO_STATUS_INVALID_DASH
AthensCairoDefs -> #CAIRO_STATUS_INVALID_DSC_COMMENT
AthensCairoDefs -> #CAIRO_STATUS_INVALID_FORMAT
AthensCairoDefs -> #CAIRO_STATUS_INVALID_INDEX
AthensCairoDefs -> #CAIRO_STATUS_INVALID_MATRIX
AthensCairoDefs -> #CAIRO_STATUS_INVALID_PATH_DATA
AthensCairoDefs -> #CAIRO_STATUS_INVALID_POP_GROUP
AthensCairoDefs -> #CAIRO_STATUS_INVALID_RESTORE
AthensCairoDefs -> #CAIRO_STATUS_INVALID_SIZE
AthensCairoDefs -> #CAIRO_STATUS_INVALID_SLANT
AthensCairoDefs -> #CAIRO_STATUS_INVALID_STATUS
AthensCairoDefs -> #CAIRO_STATUS_INVALID_STRIDE
AthensCairoDefs -> #CAIRO_STATUS_INVALID_STRING
AthensCairoDefs -> #CAIRO_STATUS_INVALID_VISUAL
AthensCairoDefs -> #CAIRO_STATUS_INVALID_WEIGHT
AthensCairoDefs -> #CAIRO_STATUS_LAST_STATUS
AthensCairoDefs -> #CAIRO_STATUS_NEGATIVE_COUNT
AthensCairoDefs -> #CAIRO_STATUS_NO_CURRENT_POINT
AthensCairoDefs -> #CAIRO_STATUS_NO_MEMORY
AthensCairoDefs -> #CAIRO_STATUS_NULL_POINTER
AthensCairoDefs -> #CAIRO_STATUS_PATTERN_TYPE_MISMATCH
AthensCairoDefs -> #CAIRO_STATUS_READ_ERROR
AthensCairoDefs -> #CAIRO_STATUS_SUCCESS
AthensCairoDefs -> #CAIRO_STATUS_SURFACE_FINISHED
AthensCairoDefs -> #CAIRO_STATUS_SURFACE_TYPE_MISMATCH

AthensCairoDefs -> #CAIRO_STATUS_TEMP_FILE_ERROR
AthensCairoDefs -> #CAIRO_STATUS_USER_FONT_ERROR
AthensCairoDefs -> #CAIRO_STATUS_USER_FONT_IMMUTABLE
AthensCairoDefs -> #CAIRO_STATUS_USER_FONT_NOT_IMPLEMENTED
AthensCairoDefs -> #CAIRO_STATUS_WRITE_ERROR
AthensCairoDefs -> #CAIRO_SUBPIXEL_ORDER_BGR
AthensCairoDefs -> #CAIRO_SUBPIXEL_ORDER_DEFAULT
AthensCairoDefs -> #CAIRO_SUBPIXEL_ORDER_RGB
AthensCairoDefs -> #CAIRO_SUBPIXEL_ORDER_VBGR
AthensCairoDefs -> #CAIRO_SUBPIXEL_ORDER_VRGB
AthensCairoDefs -> #cairo_font_slant_t
AthensCairoDefs -> #cairo_font_type_t
AthensCairoDefs -> #cairo_font_weight_t
AthensCairoDefs -> #cairo_line_cap_t
AthensCairoDefs -> #cairo_line_join_t
AthensCairoDefs -> #cairo_operator_t
AthensCairoDefs -> #cairo_pattern_t
AthensCairoDefs -> #cairo_status_t
AthensCairoDefs -> #cairo_surface_t
AthensCairoDefs -> #cairo_t
AthensCairoDefs -> #cairo_text_extents_t
AthensCurveFlattener -> #CollinearityEps
AthensCurveFlattener -> #CurveAngleToleranceEpsilon
AthensCurveFlattener -> #CurveCollinearityEpsilon
AthensCurveFlattener -> #CurveDistanceEpsilon
AthensCurveFlattener -> #SubdivisionLimit
AthensPathBuilder -> #ZeroPoint
BalloonEngineConstants -> #BEAaLevelIndex
BalloonEngineConstants -> #BEBalloonEngineSize
BalloonEngineConstants -> #BEBitBltIndex
BalloonEngineConstants -> #BEClipRectIndex
BalloonEngineConstants -> #BEColorTransformIndex
BalloonEngineConstants -> #BEDeferredIndex
BalloonEngineConstants -> #BEDestOffsetIndex
BalloonEngineConstants -> #BEEdgeTransformIndex
BalloonEngineConstants -> #BEExternalsIndex
BalloonEngineConstants -> #BEFormsIndex
BalloonEngineConstants -> #BEPostFlushNeededIndex
BalloonEngineConstants -> #BESpanIndex
BalloonEngineConstants -> #BEWorkBufferIndex
BalloonEngineConstants -> #ETBalloonEdgeDataSize
BalloonEngineConstants -> #ETIndexIndex
BalloonEngineConstants -> #ETLinesIndex
BalloonEngineConstants -> #ETSourceIndex
BalloonEngineConstants -> #ETXValueIndex
BalloonEngineConstants -> #ETYValueIndex
BalloonEngineConstants -> #ETZValueIndex
BalloonEngineConstants -> #FTBalloonFillDataSize
BalloonEngineConstants -> #FTDestFormIndex
BalloonEngineConstants -> #FTIndexIndex
BalloonEngineConstants -> #FTMaxXIndex
BalloonEngineConstants -> #FTMinXIndex
BalloonEngineConstants -> #FTSourceIndex
BalloonEngineConstants -> #FTYValueIndex
BalloonEngineConstants -> #GBBaseSize
BalloonEngineConstants -> #GBBitmapDepth
BalloonEngineConstants -> #GBBitmapHeight
BalloonEngineConstants -> #GBBitmapRaster
BalloonEngineConstants -> #GBBitmapSize
BalloonEngineConstants -> #GBBitmapWidth
BalloonEngineConstants -> #GBColormapOffset
BalloonEngineConstants -> #GBColormapSize
BalloonEngineConstants -> #GBEndX
BalloonEngineConstants -> #GBEndY
BalloonEngineConstants -> #GBFinalX
BalloonEngineConstants -> #GBMBaseSize
BalloonEngineConstants -> #GBTileFlag
BalloonEngineConstants -> #GBUpdateDDX
BalloonEngineConstants -> #GBUpdateDDY
BalloonEngineConstants -> #GBUpdateDX
BalloonEngineConstants -> #GBUpdateDY
BalloonEngineConstants -> #GBUpdateData
BalloonEngineConstants -> #GBUpdateX
BalloonEngineConstants -> #GBUpdateY
BalloonEngineConstants -> #GBViaX
BalloonEngineConstants -> #GBViaY
BalloonEngineConstants -> #GBWideEntry
BalloonEngineConstants -> #GBWideExit
BalloonEngineConstants -> #GBWideExtent
BalloonEngineConstants -> #GBWideFill
BalloonEngineConstants -> #GBWideSize
BalloonEngineConstants -> #GBWideUpdateData
BalloonEngineConstants -> #GBWideWidth
BalloonEngineConstants -> #GEBaseEdgeSize
BalloonEngineConstants -> #GEBaseFillSize
BalloonEngineConstants -> #GEEdgeClipFlag
BalloonEngineConstants -> #GEEdgeFillsInvalid
BalloonEngineConstants -> #GEFillIndexLeft
BalloonEngineConstants -> #GEFillIndexRight
BalloonEngineConstants -> #GENumLines
BalloonEngineConstants -> #GEObjectIndex
BalloonEngineConstants -> #GEObjectLength
BalloonEngineConstants -> #GEObjectType
BalloonEngineConstants -> #GEObjectUnused
BalloonEngineConstants -> #GEPrimitiveBezier
BalloonEngineConstants -> #GEPrimitiveClippedBitmapFill
BalloonEngineConstants -> #GEPrimitiveEdge
BalloonEngineConstants -> #GEPrimitiveEdgeMask
BalloonEngineConstants -> #GEPrimitiveFill
BalloonEngineConstants -> #GEPrimitiveFillMask
BalloonEngineConstants -> #GEPrimitiveLine
BalloonEngineConstants -> #GEPrimitiveLinearGradientFill
BalloonEngineConstants -> #GEPrimitiveRadialGradientFill
BalloonEngineConstants -> #GEPrimitiveRepeatedBitmapFill
BalloonEngineConstants -> #GEPrimitiveTypeMask
BalloonEngineConstants -> #GEPrimitiveUnknown
BalloonEngineConstants -> #GEPrimitiveWide
BalloonEngineConstants -> #GEPrimitiveWideBezier
BalloonEngineConstants -> #GEPrimitiveWideEdge
BalloonEngineConstants -> #GEPrimitiveWideLine
BalloonEngineConstants -> #GEPrimitiveWideMask
BalloonEngineConstants -> #GEStateAddingFromGET
BalloonEngineConstants -> #GEStateBlitBuffer
BalloonEngineConstants -> #GEStateCompleted
BalloonEngineConstants -> #GEStateScanningAET
BalloonEngineConstants -> #GEStateUnlocked
BalloonEngineConstants -> #GEStateUpdateEdges
BalloonEngineConstants -> #GEStateWaitingChange
BalloonEngineConstants -> #GEStateWaitingForEdge

BalloonEngineConstants -> #GEStateWaitingForFill
BalloonEngineConstants -> #GEXValue
BalloonEngineConstants -> #GEYValue
BalloonEngineConstants -> #GEZValue
BalloonEngineConstants -> #GErrorAETEntry
BalloonEngineConstants -> #GErrorBadState
BalloonEngineConstants -> #GErrorFillEntry
BalloonEngineConstants -> #GErrorGETEntry
BalloonEngineConstants -> #GErrorNeedFlush
BalloonEngineConstants -> #GErrorNoMoreSpace
BalloonEngineConstants -> #GFDirectionX
BalloonEngineConstants -> #GFDirectionY
BalloonEngineConstants -> #GFNormalX
BalloonEngineConstants -> #GFNormalY
BalloonEngineConstants -> #GFOriginX
BalloonEngineConstants -> #GFOriginY
BalloonEngineConstants -> #GFRampLength
BalloonEngineConstants -> #GFRampOffset
BalloonEngineConstants -> #GGBaseSize
BalloonEngineConstants -> #GLBaseSize
BalloonEngineConstants -> #GLEndX
BalloonEngineConstants -> #GLEndY
BalloonEngineConstants -> #GLError
BalloonEngineConstants -> #GLErrorAdjDown
BalloonEngineConstants -> #GLErrorAdjUp
BalloonEngineConstants -> #GLWideEntry
BalloonEngineConstants -> #GLWideExit
BalloonEngineConstants -> #GLWideExtent
BalloonEngineConstants -> #GLWideFill
BalloonEngineConstants -> #GLWideSize
BalloonEngineConstants -> #GLWideWidth
BalloonEngineConstants -> #GLXDirection
BalloonEngineConstants -> #GLXIncrement
BalloonEngineConstants -> #GLYDirection
BalloonEngineConstants -> #GWAAColorMask
BalloonEngineConstants -> #GWAAColorShift
BalloonEngineConstants -> #GWAAHalfPixel
BalloonEngineConstants -> #GWAALevel
BalloonEngineConstants -> #GWAAScanMask
BalloonEngineConstants -> #GWAAShift
BalloonEngineConstants -> #GWAETStart
BalloonEngineConstants -> #GWAETUsed
BalloonEngineConstants -> #GWBezierHeightSubdivisions
BalloonEngineConstants -> #GWBezierLineConversions
BalloonEngineConstants -> #GWBezierMonotonSubdivisions
BalloonEngineConstants -> #GWBezierOverflowSubdivisions
BalloonEngineConstants -> #GWBufferTop
BalloonEngineConstants -> #GWClearSpanBuffer
BalloonEngineConstants -> #GWClipMaxX
BalloonEngineConstants -> #GWClipMaxY
BalloonEngineConstants -> #GWClipMinX
BalloonEngineConstants -> #GWClipMinY
BalloonEngineConstants -> #GWColorTransform
BalloonEngineConstants -> #GWCountAddAETEntry
BalloonEngineConstants -> #GWCountChangeAETEntry
BalloonEngineConstants -> #GWCountDisplaySpan
BalloonEngineConstants -> #GWCountFinishTest
BalloonEngineConstants -> #GWCountInitializing
BalloonEngineConstants -> #GWCountMergeFill
BalloonEngineConstants -> #GWCountNextAETEntry
BalloonEngineConstants -> #GWCountNextFillEntry
BalloonEngineConstants -> #GWCountNextGETEntry
BalloonEngineConstants -> #GWCurrentY
BalloonEngineConstants -> #GWCurrentZ
BalloonEngineConstants -> #GWDestOffsetX
BalloonEngineConstants -> #GWDestOffsetY
BalloonEngineConstants -> #GWEdgeTransform
BalloonEngineConstants -> #GWFillMaxX
BalloonEngineConstants -> #GWFillMaxY
BalloonEngineConstants -> #GWFillMinX
BalloonEngineConstants -> #GWFillMinY
BalloonEngineConstants -> #GWFillOffsetX
BalloonEngineConstants -> #GWFillOffsetY
BalloonEngineConstants -> #GWGETStart
BalloonEngineConstants -> #GWGETUsed
BalloonEngineConstants -> #GWHasClipShapes
BalloonEngineConstants -> #GWHasColorTransform
BalloonEngineConstants -> #GWHasEdgeTransform
BalloonEngineConstants -> #GWHeaderSize
BalloonEngineConstants -> #GWLastExportedEdge
BalloonEngineConstants -> #GWLastExportedFill
BalloonEngineConstants -> #GWLastExportedLeftX
BalloonEngineConstants -> #GWLastExportedRightX
BalloonEngineConstants -> #GWMagicIndex
BalloonEngineConstants -> #GWMagicNumber
BalloonEngineConstants -> #GWMinimalSize
BalloonEngineConstants -> #GWNeedsFlush
BalloonEngineConstants -> #GWObjStart
BalloonEngineConstants -> #GWObjUsed
BalloonEngineConstants -> #GWPoint1
BalloonEngineConstants -> #GWPoint2
BalloonEngineConstants -> #GWPoint3
BalloonEngineConstants -> #GWPoint4
BalloonEngineConstants -> #GWPointListFirst
BalloonEngineConstants -> #GWSize
BalloonEngineConstants -> #GWSpanEnd
BalloonEngineConstants -> #GWSpanEndAA
BalloonEngineConstants -> #GWSpanSize
BalloonEngineConstants -> #GWSpanStart
BalloonEngineConstants -> #GWState
BalloonEngineConstants -> #GWStopReason
BalloonEngineConstants -> #GWTimeAddAETEntry
BalloonEngineConstants -> #GWTimeChangeAETEntry
BalloonEngineConstants -> #GWTimeDisplaySpan
BalloonEngineConstants -> #GWTimeFinishTest
BalloonEngineConstants -> #GWTimeInitializing
BalloonEngineConstants -> #GWTimeMergeFill
BalloonEngineConstants -> #GWTimeNextAETEntry
BalloonEngineConstants -> #GWTimeNextFillEntry
BalloonEngineConstants -> #GWTimeNextGETEntry
Base64MimeConverter -> #FromCharTable
Base64MimeConverter -> #ToCharTable
ByteString -> #NonAsciiMap
ByteTextConverter -> #byteToUnicode
ByteTextConverter -> #unicodeToByte
Categorizer -> #Default
Categorizer -> #NullCategory
Character -> #CharacterTable
Character -> #DigitValues

CharacterScanner -> #ColumnBreakStopConditions
CharacterScanner -> #CompositionStopConditions
CharacterScanner -> #DefaultStopConditions
CharacterScanner -> #MeasuringStopConditions
CharacterScanner -> #PaddedSpaceCondition
CharacterSet -> #CrLf
ChronologyConstants -> #DayNames
ChronologyConstants -> #DaysInMonth
ChronologyConstants -> #MicrosecondsInDay
ChronologyConstants -> #MonthNames
ChronologyConstants -> #NanosInMillisecond
ChronologyConstants -> #NanosInSecond
ChronologyConstants -> #SecondsInDay
ChronologyConstants -> #SecondsInHour
ChronologyConstants -> #SecondsInMinute
ChronologyConstants -> #SqueakEpoch
Color -> #BlueShift
Color -> #ColorRegistry
Color -> #ComponentMask
Color -> #ComponentMax
Color -> #GrayToIndexMap
Color -> #GreenShift
Color -> #HalfComponentMask
Color -> #IndexedColors
Color -> #RedShift
ColorPresenterMorph -> #HatchForm
CombinedChar -> #Compositions
CombinedChar -> #Decompositions
CombinedChar -> #Diacriticals
CompiledMethod -> #LargeFrame
CompiledMethod -> #SmallFrame
ContextPart -> #PrimitiveFailToken
ContextPart -> #SpecialPrimitiveSimulators
ContextPart -> #TryNamedPrimitiveTemplateMethod
CornerRounder -> #CR0
CornerRounder -> #CR1
CornerRounder -> #CR2
Cursor -> #BlankCursor
Cursor -> #BottomLeftCursor
Cursor -> #BottomRightCursor
Cursor -> #CornerCursor
Cursor -> #CrossHairCursor
Cursor -> #CurrentCursor
Cursor -> #DownCursor
Cursor -> #MarkerCursor
Cursor -> #MenuCursor
Cursor -> #MoveCursor
Cursor -> #NormalCursor
Cursor -> #OriginCursor
Cursor -> #OverEditableText
Cursor -> #ReadCursor
Cursor -> #ResizeLeftCursor
Cursor -> #ResizeTopCursor
Cursor -> #ResizeTopLeftCursor
Cursor -> #ResizeTopRightCursor
Cursor -> #RightArrowCursor
Cursor -> #SquareCursor
Cursor -> #TargetCursor
Cursor -> #TopLeftCursor
Cursor -> #TopRightCursor
Cursor -> #UpCursor
Cursor -> #WaitCursor
Cursor -> #WebLinkCursor
Cursor -> #WriteCursor
Cursor -> #XeqCursor
Decompiler -> #ArgumentFlag
Decompiler -> #CascadeFlag
Decompiler -> #CaseFlag
Decompiler -> #IfNilFlag
DigitalSignatureAlgorithm -> #HighBitOfByte
DigitalSignatureAlgorithm -> #SmallPrimes
DisplayMedium -> #HighLightBitmaps
EventSensorConstants -> #BlueButtonBit
EventSensorConstants -> #CommandKeyBit
EventSensorConstants -> #CtrlKeyBit
EventSensorConstants -> #EventKeyChar
EventSensorConstants -> #EventKeyDown
EventSensorConstants -> #EventKeyUp
EventSensorConstants -> #EventTypeDragDropFiles
EventSensorConstants -> #EventTypeKeyboard
EventSensorConstants -> #EventTypeMenu
EventSensorConstants -> #EventTypeMouse
EventSensorConstants -> #EventTypeNone
EventSensorConstants -> #EventTypeWindow
EventSensorConstants -> #OptionKeyBit
EventSensorConstants -> #RedButtonBit
EventSensorConstants -> #ShiftKeyBit
EventSensorConstants -> #WindowEventActivated
EventSensorConstants -> #WindowEventClose
EventSensorConstants -> #WindowEventIconise
EventSensorConstants -> #WindowEventMetricChange
EventSensorConstants -> #WindowEventPaint
EventSensorConstants -> #YellowButtonBit
FLLargeIdentityHashedCollection -> #PermutationMap
FT2Constants -> #LoadCropBitmap
FT2Constants -> #LoadDefault
FT2Constants -> #LoadForceAutohint
FT2Constants -> #LoadIgnoreGlobalAdvanceWidth
FT2Constants -> #LoadIgnoreTransform
FT2Constants -> #LoadLinearDesign
FT2Constants -> #LoadMonochrome
FT2Constants -> #LoadNoAutohint
FT2Constants -> #LoadNoBitmap
FT2Constants -> #LoadNoHinting
FT2Constants -> #LoadNoRecurse
FT2Constants -> #LoadNoScale
FT2Constants -> #LoadPedantic
FT2Constants -> #LoadRender
FT2Constants -> #LoadSbitsOnly
FT2Constants -> #LoadTargetLCD
FT2Constants -> #LoadTargetLCDV
FT2Constants -> #LoadTargetLight
FT2Constants -> #LoadTargetMono
FT2Constants -> #LoadTargetNormal
FT2Constants -> #LoadVerticalLayout
FT2Constants -> #PixelModeGray
FT2Constants -> #PixelModeGray2
FT2Constants -> #PixelModeGray4
FT2Constants -> #PixelModeLCD

FT2Constants -> #PixelModeLCDV
FT2Constants -> #PixelModeMono
FT2Constants -> #PixelModeNone
FT2Constants -> #RenderModeLCD
FT2Constants -> #RenderModeLCDV
FT2Constants -> #RenderModeLight
FT2Constants -> #RenderModeMono
FT2Constants -> #RenderModeNormal
FT2Constants -> #StyleFlagBold
FT2Constants -> #StyleFlagItalic
FastInflateStream -> #DistanceMap
FastInflateStream -> #FixedDistTable
FastInflateStream -> #FixedLitTable
FastInflateStream -> #LiteralLengthMap
Float -> #E
Float -> #Epsilon
Float -> #Halfpi
Float -> #Infinity
Float -> #Ln10
Float -> #Ln2
Float -> #MaxVal
Float -> #MaxValLn
Float -> #MinValLogBase2
Float -> #NaN
Float -> #NegativeInfinity
Float -> #NegativeZero
Float -> #Pi
Float -> #RadiansPerDegree
Float -> #Sqrt2
Float -> #ThreePi
Float -> #Twopi
FormCanvas -> #TranslucentPatterns
FreeTypeCacheConstants -> #FreeTypeCacheGlyph
FreeTypeCacheConstants -> #FreeTypeCacheGlyphLCD
FreeTypeCacheConstants -> #FreeTypeCacheGlyphMono
FreeTypeCacheConstants -> #FreeTypeCacheLinearWidth
FreeTypeCacheConstants -> #FreeTypeCacheWidth
FreeTypeNameParser -> #italicNames
FreeTypeNameParser -> #normalNames
FreeTypeNameParser -> #obliqueNames
FreeTypeNameParser -> #stretchNames
FreeTypeNameParser -> #weightNames
GIFReadWriter -> #Extension
GIFReadWriter -> #ImageSeparator
GIFReadWriter -> #Terminator
GZipConstants -> #GZipAsciiFlag
GZipConstants -> #GZipCommentFlag
GZipConstants -> #GZipContinueFlag
GZipConstants -> #GZipDeflated
GZipConstants -> #GZipEncryptFlag
GZipConstants -> #GZipExtraField
GZipConstants -> #GZipMagic
GZipConstants -> #GZipNameFlag
GZipConstants -> #GZipReservedFlags
HashTableSizes -> #sizes
IRBytecodeGenerator -> #BytecodeTable
IRBytecodeGenerator -> #Bytecodes
IRBytecodeGenerator -> #SpecialConstants
IRBytecodeGenerator -> #SpecialSelectors
ISOLanguageDefinition -> #ISO2Countries
ISOLanguageDefinition -> #ISO2Table
ISOLanguageDefinition -> #ISO3Countries
ISOLanguageDefinition -> #ISO3Table
IconicButton -> #DefaultGraphics
ImageMorph -> #DefaultForm
InflateStream -> #BlockProceedBit
InflateStream -> #BlockTypes
InflateStream -> #FixedDistCodes
InflateStream -> #FixedLitCodes
InflateStream -> #MaxBits
InflateStream -> #StateNewBlock
InflateStream -> #StateNoMoreData
InputEventSensor -> #ButtonDecodeTable
InstructionStream -> #SpecialConstants
JPEGHuffmanTable -> #BitBufferSize
JPEGHuffmanTable -> #Lookahead
JPEGReadStream -> #MaxBits
JPEGReadWriter -> #ConstBits
JPEGReadWriter -> #DCTK1
JPEGReadWriter -> #DCTK2
JPEGReadWriter -> #DCTK3
JPEGReadWriter -> #DCTK4
JPEGReadWriter -> #DCTSize
JPEGReadWriter -> #DCTSize2
JPEGReadWriter -> #DitherMasks
JPEGReadWriter -> #FIXn0n298631336
JPEGReadWriter -> #FIXn0n34414
JPEGReadWriter -> #FIXn0n390180644
JPEGReadWriter -> #FIXn0n541196100
JPEGReadWriter -> #FIXn0n71414
JPEGReadWriter -> #FIXn0n765366865
JPEGReadWriter -> #FIXn0n899976223
JPEGReadWriter -> #FIXn1n175875602
JPEGReadWriter -> #FIXn1n40200
JPEGReadWriter -> #FIXn1n501321110
JPEGReadWriter -> #FIXn1n77200
JPEGReadWriter -> #FIXn1n847759065
JPEGReadWriter -> #FIXn1n961570560
JPEGReadWriter -> #FIXn2n053119869
JPEGReadWriter -> #FIXn2n562915447
JPEGReadWriter -> #FIXn3n072711026
JPEGReadWriter -> #FloatSampleOffset
JPEGReadWriter -> #HuffmanTableSize
JPEGReadWriter -> #JFIFMarkerParser
JPEGReadWriter -> #JPEGNaturalOrder
JPEGReadWriter -> #MaxSample
JPEGReadWriter -> #Pass1Bits
JPEGReadWriter -> #Pass1Div
JPEGReadWriter -> #Pass2Div
JPEGReadWriter -> #QTableScaleFactor
JPEGReadWriter -> #QuantizationTableSize
JPEGReadWriter -> #SampleOffset
KMSingleKeyCombination -> #specialKeys
Key -> #KeyTable
Key -> #MacosVirtualKeyTable
Key -> #UnixVirtualKeyTable
Key -> #WindowsVirtualKeyTable
KomitClassNode -> #addedClassIcon
KomitClassNode -> #deletedClassIcon

KomitClassNode -> #modifiedClassIcon
KomitterUI -> #manageRemotesIcon
Latin1 -> #rightHalfSequence
MCDataStream -> #TypeMap
MCFileTreeStCypressWriter -> #specials
MD5NonPrimitive -> #ABCDTable
MD5NonPrimitive -> #IndexTable
MD5NonPrimitive -> #ShiftTable
MD5NonPrimitive -> #SinTable
MailAddressTokenizer -> #CSNonAtom
MailAddressTokenizer -> #CSNonSeparators
MailAddressTokenizer -> #CSParens
MailAddressTokenizer -> #CSSpecials
MenuItemMorph -> #SubMenuMarker
MessageNode -> #MacroEmitters
MessageNode -> #MacroPrinters
MessageNode -> #MacroSelectors
MessageNode -> #MacroSizers
MessageNode -> #MacroTransformers
MessageNode -> #StdTypers
MessageNode -> #ThenFlag
MetacelloVersionValidator -> #reasonCodeDescriptions
Morph -> #EmptyArray
MultiByteFileStream -> #Cr
MultiByteFileStream -> #CrLf
MultiByteFileStream -> #Lf
MultiByteFileStream -> #LineEndStrings
MultiByteFileStream -> #LookAheadCount
NBFFICallout -> #CustomErrorCodes
NBFFICallout -> #CustomErrorMessages
NBFFICallout -> #TypeAliases
NBInterpreterProxy -> #CogFunctions
NBInterpreterProxy -> #Functions
NBMacConstants -> #MAP_ANON
NBMacConstants -> #MAP_COPY
NBMacConstants -> #MAP_FAILED
NBMacConstants -> #MAP_FILE
NBMacConstants -> #MAP_FIXED
NBMacConstants -> #MAP_HASSEMAPHORE
NBMacConstants -> #MAP_NOCACHE
NBMacConstants -> #MAP_NOEXTEND
NBMacConstants -> #MAP_NORESERVE
NBMacConstants -> #MAP_PRIVATE
NBMacConstants -> #MAP_RENAME
NBMacConstants -> #MAP_RESERVED0080
NBMacConstants -> #MAP_SHARED
NBMacConstants -> #PROT_EXEC
NBMacConstants -> #PROT_NONE
NBMacConstants -> #PROT_READ
NBMacConstants -> #PROT_WRITE
NBMacConstants -> #RTLD_DEFAULT
NBMacConstants -> #RTLD_FIRST
NBMacConstants -> #RTLD_GLOBAL
NBMacConstants -> #RTLD_LAZY
NBMacConstants -> #RTLD_LOCAL
NBMacConstants -> #RTLD_MAIN_ONLY
NBMacConstants -> #RTLD_NEXT
NBMacConstants -> #RTLD_NODELETE
NBMacConstants -> #RTLD_NOLOAD
NBMacConstants -> #RTLD_NOW
NBMacConstants -> #RTLD_SELF
NBUnixConstants -> #MAP_32BIT
NBUnixConstants -> #MAP_ANON
NBUnixConstants -> #MAP_ANONYMOUS
NBUnixConstants -> #MAP_DENYWRITE
NBUnixConstants -> #MAP_EXECUTABLE
NBUnixConstants -> #MAP_FAILED
NBUnixConstants -> #MAP_FILE
NBUnixConstants -> #MAP_FIXED
NBUnixConstants -> #MAP_GROWSDOWN
NBUnixConstants -> #MAP_LOCKED
NBUnixConstants -> #MAP_NONBLOCK
NBUnixConstants -> #MAP_NORESERVE
NBUnixConstants -> #MAP_POPULATE
NBUnixConstants -> #MAP_PRIVATE
NBUnixConstants -> #MAP_SHARED
NBUnixConstants -> #MAP_STACK
NBUnixConstants -> #MAP_TYPE
NBUnixConstants -> #PROT_EXEC
NBUnixConstants -> #PROT_GROWSDOWN
NBUnixConstants -> #PROT_GROWSUP
NBUnixConstants -> #PROT_NONE
NBUnixConstants -> #PROT_READ
NBUnixConstants -> #PROT_WRITE
NBUnixConstants -> #RTLD_BINDING_MASK
NBUnixConstants -> #RTLD_DEEPBIND
NBUnixConstants -> #RTLD_DEFAULT
NBUnixConstants -> #RTLD_GLOBAL
NBUnixConstants -> #RTLD_LAZY
NBUnixConstants -> #RTLD_LOCAL
NBUnixConstants -> #RTLD_NEXT
NBUnixConstants -> #RTLD_NODELETE
NBUnixConstants -> #RTLD_NOLOAD
NBUnixConstants -> #RTLD_NOW
NBWinConstants -> #ABOVE_NORMAL_PRIORITY_CLASS
NBWinConstants -> #ACCESS_SYSTEM_SECURITY
NBWinConstants -> #ACE_INHERITED_OBJECT_TYPE_PRESENT
NBWinConstants -> #ACE_OBJECT_TYPE_PRESENT
NBWinConstants -> #APPLICATION_ERROR_MASK
NBWinConstants -> #BELOW_NORMAL_PRIORITY_CLASS
NBWinConstants -> #CREATE_FORCEDOS
NBWinConstants -> #CREATE_NEW_CONSOLE
NBWinConstants -> #CREATE_NEW_PROCESS_GROUP
NBWinConstants -> #CREATE_SEPARATE_WOW_VDM
NBWinConstants -> #CREATE_SHARED_WOW_VDM
NBWinConstants -> #CREATE_SUSPENDED
NBWinConstants -> #CREATE_UNICODE_ENVIRONMENT
NBWinConstants -> #CS_BYTEALIGNCLIENT
NBWinConstants -> #CS_BYTEALIGNWINDOW
NBWinConstants -> #CS_CLASSDC
NBWinConstants -> #CS_DBLCLKS
NBWinConstants -> #CS_DROPSHADOW
NBWinConstants -> #CS_GLOBALCLASS
NBWinConstants -> #CS_HREDRAW
NBWinConstants -> #CS_IME
NBWinConstants -> #CS_NOCLOSE
NBWinConstants -> #CS_OWNDC
NBWinConstants -> #CS_PARENTDC
NBWinConstants -> #CS_SAVEBITS

NBWinConstants -> #CS_VREDRAW
NBWinConstants -> #CW_USEDEFAULT
NBWinConstants -> #DEBUG_ONLY_THIS_PROCESS
NBWinConstants -> #DEBUG_PROCESS
NBWinConstants -> #DETACHED_PROCESS
NBWinConstants -> #DRIVE_CDROM
NBWinConstants -> #DRIVE_FIXED
NBWinConstants -> #DRIVE_NO_ROOT_DIR
NBWinConstants -> #DRIVE_RAMDISK
NBWinConstants -> #DRIVE_REMOTE
NBWinConstants -> #DRIVE_REMOVABLE
NBWinConstants -> #DRIVE_UNKNOWN
NBWinConstants -> #ERROR_SEVERITY_ERROR
NBWinConstants -> #ERROR_SEVERITY_INFORMATIONAL
NBWinConstants -> #ERROR_SEVERITY_SUCCESS
NBWinConstants -> #ERROR_SEVERITY_WARNING
NBWinConstants -> #GWL_EXSTYLE
NBWinConstants -> #GWL_HINSTANCE
NBWinConstants -> #GWL_HWNDPARENT
NBWinConstants -> #GWL_ID
NBWinConstants -> #GWL_STYLE
NBWinConstants -> #GWL_USERDATA
NBWinConstants -> #GWL_WNDPROC
NBWinConstants -> #GW_CHILD
NBWinConstants -> #GW_ENABLEDPOPUP
NBWinConstants -> #GW_HWNDFIRST
NBWinConstants -> #GW_HWNDLAST
NBWinConstants -> #GW_HWNDNEXT
NBWinConstants -> #GW_HWNDPREV
NBWinConstants -> #GW_OWNER
NBWinConstants -> #HEAP_CREATE_ENABLE_EXECUTE
NBWinConstants -> #HEAP_GENERATE_EXCEPTIONS
NBWinConstants -> #HEAP_NO_SERIALIZE
NBWinConstants -> #HEAP_REALLOC_IN_PLACE_ONLY
NBWinConstants -> #HEAP_ZERO_MEMORY
NBWinConstants -> #HIGH_PRIORITY_CLASS
NBWinConstants -> #IDABORT
NBWinConstants -> #IDCANCEL
NBWinConstants -> #IDCONTINUE
NBWinConstants -> #IDIGNORE
NBWinConstants -> #IDLE_PRIORITY_CLASS
NBWinConstants -> #IDNO
NBWinConstants -> #IDOK
NBWinConstants -> #IDRETRY
NBWinConstants -> #IDTRYAGAIN
NBWinConstants -> #IDYES
NBWinConstants -> #MB_ABORTRETRYIGNORE
NBWinConstants -> #MB_APPLMODAL
NBWinConstants -> #MB_CANCELTRYCONTINUE
NBWinConstants -> #MB_DEFAULT_DESKTOP_ONLY
NBWinConstants -> #MB_DEFBUTTON1
NBWinConstants -> #MB_DEFBUTTON2
NBWinConstants -> #MB_DEFBUTTON3
NBWinConstants -> #MB_DEFBUTTON4
NBWinConstants -> #MB_HELP
NBWinConstants -> #MB_ICONASTERISK
NBWinConstants -> #MB_ICONERROR
NBWinConstants -> #MB_ICONEXCLAMATION
NBWinConstants -> #MB_ICONHAND
NBWinConstants -> #MB_ICONINFORMATION
NBWinConstants -> #MB_ICONQUESTION
NBWinConstants -> #MB_ICONSTOP
NBWinConstants -> #MB_ICONWARNING
NBWinConstants -> #MB_OK
NBWinConstants -> #MB_OKCANCEL
NBWinConstants -> #MB_RETRYCANCEL
NBWinConstants -> #MB_RIGHT
NBWinConstants -> #MB_RTLREADING
NBWinConstants -> #MB_SERVICE_NOTIFICATION
NBWinConstants -> #MB_SETFOREGROUND
NBWinConstants -> #MB_SYSTEMMODAL
NBWinConstants -> #MB_TASKMODAL
NBWinConstants -> #MB_TOPMOST
NBWinConstants -> #MB_YESNO
NBWinConstants -> #MB_YESNOCANCEL
NBWinConstants -> #NORMAL_PRIORITY_CLASS
NBWinConstants -> #PFD_DEPTH_DONTCARE
NBWinConstants -> #PFD_DOUBLEBUFFER
NBWinConstants -> #PFD_DOUBLEBUFFER_DONTCARE
NBWinConstants -> #PFD_DRAW_TO_BITMAP
NBWinConstants -> #PFD_DRAW_TO_WINDOW
NBWinConstants -> #PFD_GENERIC_ACCELERATED
NBWinConstants -> #PFD_GENERIC_FORMAT
NBWinConstants -> #PFD_MAIN_PLANE
NBWinConstants -> #PFD_NEED_PALETTE
NBWinConstants -> #PFD_NEED_SYSTEM_PALETTE
NBWinConstants -> #PFD_OVERLAY_PLANE
NBWinConstants -> #PFD_STEREO
NBWinConstants -> #PFD_STEREO_DONTCARE
NBWinConstants -> #PFD_SUPPORT_DIRECTDRAW
NBWinConstants -> #PFD_SUPPORT_GDI
NBWinConstants -> #PFD_SUPPORT_OPENGL
NBWinConstants -> #PFD_SWAP_COPY
NBWinConstants -> #PFD_SWAP_EXCHANGE
NBWinConstants -> #PFD_SWAP_LAYER_BUFFERS
NBWinConstants -> #PFD_TYPE_COLORINDEX
NBWinConstants -> #PFD_TYPE_RGBA
NBWinConstants -> #PFD_UNDERLAY_PLANE
NBWinConstants -> #REALTIME_PRIORITY_CLASS
NBWinConstants -> #SM_ARRANGE
NBWinConstants -> #SM_CLEANBOOT
NBWinConstants -> #SM_CMONITORS
NBWinConstants -> #SM_CMOUSEBUTTONS
NBWinConstants -> #SM_CXBORDER
NBWinConstants -> #SM_CXCURSOR
NBWinConstants -> #SM_CXDLGFRAME
NBWinConstants -> #SM_CXDOUBLECLK
NBWinConstants -> #SM_CXDRAG
NBWinConstants -> #SM_CXEDGE
NBWinConstants -> #SM_CXFIXEDFRAME
NBWinConstants -> #SM_CXFOCUSBORDER
NBWinConstants -> #SM_CXFRAME
NBWinConstants -> #SM_CXFULLSCREEN
NBWinConstants -> #SM_CXHSCROLL
NBWinConstants -> #SM_CXHTHUMB
NBWinConstants -> #SM_CXICON
NBWinConstants -> #SM_CXICONSPACING
NBWinConstants -> #SM_CXMAXIMIZED
NBWinConstants -> #SM_CXMAXTRACK

NBWinConstants -> #SM_CXMENUCHECK
NBWinConstants -> #SM_CXMENUSIZE
NBWinConstants -> #SM_CXMIN
NBWinConstants -> #SM_CXMINIMIZED
NBWinConstants -> #SM_CXMINSPACING
NBWinConstants -> #SM_CXMINTRACK
NBWinConstants -> #SM_CXPADDEDBORDER
NBWinConstants -> #SM_CXSCREEN
NBWinConstants -> #SM_CXSIZE
NBWinConstants -> #SM_CXSIZEFRAME
NBWinConstants -> #SM_CXSMICON
NBWinConstants -> #SM_CXSMSIZE
NBWinConstants -> #SM_CXVIRTUALSCREEN
NBWinConstants -> #SM_CXVSCROLL
NBWinConstants -> #SM_CYBORDER
NBWinConstants -> #SM_CYCAPTION
NBWinConstants -> #SM_CYCURSOR
NBWinConstants -> #SM_CYDLGFRAME
NBWinConstants -> #SM_CYDOUBLECLK
NBWinConstants -> #SM_CYDRAG
NBWinConstants -> #SM_CYEDGE
NBWinConstants -> #SM_CYFIXEDFRAME
NBWinConstants -> #SM_CYFOCUSBORDER
NBWinConstants -> #SM_CYFRAME
NBWinConstants -> #SM_CYFULLSCREEN
NBWinConstants -> #SM_CYHSCROLL
NBWinConstants -> #SM_CYICON
NBWinConstants -> #SM_CYICONSPACING
NBWinConstants -> #SM_CYKANJIWINDOW
NBWinConstants -> #SM_CYMAXIMIZED
NBWinConstants -> #SM_CYMAXTRACK
NBWinConstants -> #SM_CYMENU
NBWinConstants -> #SM_CYMENUCHECK
NBWinConstants -> #SM_CYMENUSIZE
NBWinConstants -> #SM_CYMIN
NBWinConstants -> #SM_CYMINIMIZED
NBWinConstants -> #SM_CYMINSPACING
NBWinConstants -> #SM_CYMINTRACK
NBWinConstants -> #SM_CYSCREEN
NBWinConstants -> #SM_CYSIZE
NBWinConstants -> #SM_CYSIZEFRAME
NBWinConstants -> #SM_CYSMCAPTION
NBWinConstants -> #SM_CYSMICON
NBWinConstants -> #SM_CYSMSIZE
NBWinConstants -> #SM_CYVIRTUALSCREEN
NBWinConstants -> #SM_CYVSCROLL
NBWinConstants -> #SM_CYVTHUMB
NBWinConstants -> #SM_DBCSENABLED
NBWinConstants -> #SM_DEBUG
NBWinConstants -> #SM_DIGITIZER
NBWinConstants -> #SM_IMMENABLED
NBWinConstants -> #SM_MAXIMUMTOUCHES
NBWinConstants -> #SM_MEDIACENTER
NBWinConstants -> #SM_MENUDROPALIGNMENT
NBWinConstants -> #SM_MIDEASTENABLED
NBWinConstants -> #SM_MOUSEHORIZONTALWHEELPRESENT
NBWinConstants -> #SM_MOUSEPRESENT
NBWinConstants -> #SM_MOUSEWHEELPRESENT
NBWinConstants -> #SM_NETWORK
NBWinConstants -> #SM_PENWINDOWS
NBWinConstants -> #SM_REMOTECONTROL
NBWinConstants -> #SM_REMOTESESSION
NBWinConstants -> #SM_SAMEDISPLAYFORMAT
NBWinConstants -> #SM_SECURE
NBWinConstants -> #SM_SHOWSOUNDS
NBWinConstants -> #SM_SHUTTINGDOWN
NBWinConstants -> #SM_SLOWMACHINE
NBWinConstants -> #SM_STARTER
NBWinConstants -> #SM_SWAPBUTTON
NBWinConstants -> #SM_TABLETPC
NBWinConstants -> #SM_XVIRTUALSCREEN
NBWinConstants -> #SM_YVIRTUALSCREEN
NBWinConstants -> #SPECIFIC_RIGHTS_ALL
NBWinConstants -> #STANDARD_RIGHTS_ALL
NBWinConstants -> #STANDARD_RIGHTS_EXECUTE
NBWinConstants -> #STANDARD_RIGHTS_READ
NBWinConstants -> #STANDARD_RIGHTS_REQUIRED
NBWinConstants -> #STANDARD_RIGHTS_WRITE
NBWinConstants -> #SW_FORCEMINIMIZE
NBWinConstants -> #SW_HIDE
NBWinConstants -> #SW_MAX
NBWinConstants -> #SW_MAXIMIZE
NBWinConstants -> #SW_MINIMIZE
NBWinConstants -> #SW_NORMAL
NBWinConstants -> #SW_RESTORE
NBWinConstants -> #SW_SHOW
NBWinConstants -> #SW_SHOWDEFAULT
NBWinConstants -> #SW_SHOWMAXIMIZED
NBWinConstants -> #SW_SHOWMINIMIZED
NBWinConstants -> #SW_SHOWMINNOACTIVE
NBWinConstants -> #SW_SHOWNA
NBWinConstants -> #SW_SHOWNOACTIVATE
NBWinConstants -> #SW_SHOWNORMAL
NBWinConstants -> #SYNCHRONIZE
NBWinConstants -> #THREAD_ALL_ACCESS
NBWinConstants -> #THREAD_DIRECT_IMPERSONATION
NBWinConstants -> #THREAD_GET_CONTEXT
NBWinConstants -> #THREAD_IMPERSONATE
NBWinConstants -> #THREAD_QUERY_INFORMATION
NBWinConstants -> #THREAD_SET_CONTEXT
NBWinConstants -> #THREAD_SET_INFORMATION
NBWinConstants -> #THREAD_SET_THREAD_TOKEN
NBWinConstants -> #THREAD_SUSPEND_RESUME
NBWinConstants -> #THREAD_TERMINATE
NBWinConstants -> #WM_ACTIVATEAPP
NBWinConstants -> #WM_CANCELMODE
NBWinConstants -> #WM_CHILDACTIVATE
NBWinConstants -> #WM_CLOSE
NBWinConstants -> #WM_COMPACTING
NBWinConstants -> #WM_CREATE
NBWinConstants -> #WM_DESTROY
NBWinConstants -> #WM_ENABLE
NBWinConstants -> #WM_ENTERSIZEMOVE
NBWinConstants -> #WM_EXITSIZEMOVE
NBWinConstants -> #WM_GETICON
NBWinConstants -> #WM_GETMINMAXINFO
NBWinConstants -> #WM_INPUTLANGCHANGE
NBWinConstants -> #WM_INPUTLANGCHANGEREQUEST
NBWinConstants -> #WM_MOVE

NBWinConstants -> #WM_MOVING
NBWinConstants -> #WM_NCACTIVATE
NBWinConstants -> #WM_NCCALCSIZE
NBWinConstants -> #WM_NCCREATE
NBWinConstants -> #WM_NCDESTROY
NBWinConstants -> #WM_NULL
NBWinConstants -> #WM_QUERYDRAGICON
NBWinConstants -> #WM_QUERYOPEN
NBWinConstants -> #WM_QUIT
NBWinConstants -> #WM_SHOWWINDOW
NBWinConstants -> #WM_SIZE
NBWinConstants -> #WM_SIZING
NBWinConstants -> #WM_STYLECHANGED
NBWinConstants -> #WM_STYLECHANGING
NBWinConstants -> #WM_THEMECHANGED
NBWinConstants -> #WM_USERCHANGED
NBWinConstants -> #WM_WINDOWPOSCHANGED
NBWinConstants -> #WM_WINDOWPOSCHANGING
NBWinConstants -> #WS_BORDER
NBWinConstants -> #WS_CAPTION
NBWinConstants -> #WS_CHILD
NBWinConstants -> #WS_CHILDWINDOW
NBWinConstants -> #WS_CLIPCHILDREN
NBWinConstants -> #WS_CLIPSIBLINGS
NBWinConstants -> #WS_DISABLED
NBWinConstants -> #WS_DLGFRAME
NBWinConstants -> #WS_EX_ACCEPTFILES
NBWinConstants -> #WS_EX_APPWINDOW
NBWinConstants -> #WS_EX_CLIENTEDGE
NBWinConstants -> #WS_EX_COMPOSITED
NBWinConstants -> #WS_EX_CONTEXTHELP
NBWinConstants -> #WS_EX_CONTROLPARENT
NBWinConstants -> #WS_EX_DLGMODALFRAME
NBWinConstants -> #WS_EX_LAYERED
NBWinConstants -> #WS_EX_LAYOUTRTL
NBWinConstants -> #WS_EX_LEFT
NBWinConstants -> #WS_EX_LEFTSCROLLBAR
NBWinConstants -> #WS_EX_LTRREADING
NBWinConstants -> #WS_EX_MDICHILD
NBWinConstants -> #WS_EX_NOACTIVATE
NBWinConstants -> #WS_EX_NOINHERITLAYOUT
NBWinConstants -> #WS_EX_NOPARENTNOTIFY
NBWinConstants -> #WS_EX_OVERLAPPEDWINDOW
NBWinConstants -> #WS_EX_PALETTEWINDOW
NBWinConstants -> #WS_EX_RIGHT
NBWinConstants -> #WS_EX_RIGHTSCROLLBAR
NBWinConstants -> #WS_EX_RTLREADING
NBWinConstants -> #WS_EX_STATICEDGE
NBWinConstants -> #WS_EX_TOOLWINDOW
NBWinConstants -> #WS_EX_TOPMOST
NBWinConstants -> #WS_EX_TRANSPARENT
NBWinConstants -> #WS_EX_WINDOWEDGE
NBWinConstants -> #WS_GROUP
NBWinConstants -> #WS_HSCROLL
NBWinConstants -> #WS_ICONIC
NBWinConstants -> #WS_MAXIMIZE
NBWinConstants -> #WS_MAXIMIZEBOX
NBWinConstants -> #WS_MINIMIZE
NBWinConstants -> #WS_MINIMIZEBOX
NBWinConstants -> #WS_OVERLAPPED
NBWinConstants -> #WS_OVERLAPPEDWINDOW
NBWinConstants -> #WS_POPUP
NBWinConstants -> #WS_POPUPWINDOW
NBWinConstants -> #WS_SIZEBOX
NBWinConstants -> #WS_SYSMENU
NBWinConstants -> #WS_TABSTOP
NBWinConstants -> #WS_THICKFRAME
NBWinConstants -> #WS_TILED
NBWinConstants -> #WS_TILEDWINDOW
NBWinConstants -> #WS_VISIBLE
NBWinConstants -> #WS_VSCROLL
NBWinTypes -> #ATOM
NBWinTypes -> #BOOL
NBWinTypes -> #BOOLEAN
NBWinTypes -> #BYTE
NBWinTypes -> #CALLBACK
NBWinTypes -> #CHAR
NBWinTypes -> #COLORREF
NBWinTypes -> #DWORD
NBWinTypes -> #DWORD32
NBWinTypes -> #DWORD64
NBWinTypes -> #DWORDLONG
NBWinTypes -> #DWORD_PTR
NBWinTypes -> #FLOAT
NBWinTypes -> #HACCEL
NBWinTypes -> #HALF_PTR
NBWinTypes -> #HANDLE
NBWinTypes -> #HBRUSH
NBWinTypes -> #HCOLORSPACE
NBWinTypes -> #HCONV
NBWinTypes -> #HCONVLIST
NBWinTypes -> #HCURSOR
NBWinTypes -> #HDC
NBWinTypes -> #HDDEDATA
NBWinTypes -> #HDESK
NBWinTypes -> #HDROP
NBWinTypes -> #HDWP
NBWinTypes -> #HENHMETAFILE
NBWinTypes -> #HFILE
NBWinTypes -> #HFONT
NBWinTypes -> #HGDIOBJ
NBWinTypes -> #HGLOBAL
NBWinTypes -> #HHOOK
NBWinTypes -> #HICON
NBWinTypes -> #HINSTANCE
NBWinTypes -> #HKEY
NBWinTypes -> #HKL
NBWinTypes -> #HLOCAL
NBWinTypes -> #HMENU
NBWinTypes -> #HMETAFILE
NBWinTypes -> #HMODULE
NBWinTypes -> #HMONITOR
NBWinTypes -> #HPALETTE
NBWinTypes -> #HPEN
NBWinTypes -> #HRESULT
NBWinTypes -> #HRGN
NBWinTypes -> #HRSRC
NBWinTypes -> #HSZ
NBWinTypes -> #HWINSTA

NBWinTypes -> #HWND
NBWinTypes -> #INT
NBWinTypes -> #INT32
NBWinTypes -> #INT64
NBWinTypes -> #INT_PTR
NBWinTypes -> #LANGID
NBWinTypes -> #LCID
NBWinTypes -> #LCTYPE
NBWinTypes -> #LGRPID
NBWinTypes -> #LONG
NBWinTypes -> #LONG32
NBWinTypes -> #LONG64
NBWinTypes -> #LONGLONG
NBWinTypes -> #LONG_PTR
NBWinTypes -> #LPARAM
NBWinTypes -> #LPBOOL
NBWinTypes -> #LPBYTE
NBWinTypes -> #LPCOLORREF
NBWinTypes -> #LPCSTR
NBWinTypes -> #LPCTSTR
NBWinTypes -> #LPCVOID
NBWinTypes -> #LPCWSTR
NBWinTypes -> #LPDWORD
NBWinTypes -> #LPHANDLE
NBWinTypes -> #LPINT
NBWinTypes -> #LPLONG
NBWinTypes -> #LPSTR
NBWinTypes -> #LPTCH
NBWinTypes -> #LPTSTR
NBWinTypes -> #LPVOID
NBWinTypes -> #LPWCH
NBWinTypes -> #LPWORD
NBWinTypes -> #LPWSTR
NBWinTypes -> #LRESULT
NBWinTypes -> #PBOOL
NBWinTypes -> #PBOOLEAN
NBWinTypes -> #PBYTE
NBWinTypes -> #PCHAR
NBWinTypes -> #PCSTR
NBWinTypes -> #PCTSTR
NBWinTypes -> #PCWSTR
NBWinTypes -> #PDWORD
NBWinTypes -> #PDWORD32
NBWinTypes -> #PDWORD64
NBWinTypes -> #PDWORDLONG
NBWinTypes -> #PDWORD_PTR
NBWinTypes -> #PFLOAT
NBWinTypes -> #PHALF_PTR
NBWinTypes -> #PHANDLE
NBWinTypes -> #PHKEY
NBWinTypes -> #PINT
NBWinTypes -> #PINT32
NBWinTypes -> #PINT64
NBWinTypes -> #PINT_PTR
NBWinTypes -> #PLCID
NBWinTypes -> #PLONG
NBWinTypes -> #PLONG32
NBWinTypes -> #PLONG64
NBWinTypes -> #PLONGLONG
NBWinTypes -> #PLONG_PTR
NBWinTypes -> #POINT
NBWinTypes -> #POINTER_32
NBWinTypes -> #POINTER_64
NBWinTypes -> #PSHORT
NBWinTypes -> #PSIZE_T
NBWinTypes -> #PSSIZE_T
NBWinTypes -> #PSTR
NBWinTypes -> #PTBYTE
NBWinTypes -> #PTCHAR
NBWinTypes -> #PTSTR
NBWinTypes -> #PUCHAR
NBWinTypes -> #PUHALF_PTR
NBWinTypes -> #PUINT
NBWinTypes -> #PUINT32
NBWinTypes -> #PUINT64
NBWinTypes -> #PUINT_PTR
NBWinTypes -> #PULONG
NBWinTypes -> #PULONG32
NBWinTypes -> #PULONG64
NBWinTypes -> #PULONGLONG
NBWinTypes -> #PULONG_PTR
NBWinTypes -> #PUSHORT
NBWinTypes -> #PVOID
NBWinTypes -> #PWCHAR
NBWinTypes -> #PWORD
NBWinTypes -> #PWSTR
NBWinTypes -> #RECT
NBWinTypes -> #SC_HANDLE
NBWinTypes -> #SC_LOCK
NBWinTypes -> #SERVICE_STATUS_HANDLE
NBWinTypes -> #SHORT
NBWinTypes -> #SIZE_T
NBWinTypes -> #SSIZE_T
NBWinTypes -> #TBYTE
NBWinTypes -> #TCHAR
NBWinTypes -> #UCHAR
NBWinTypes -> #UHALF_PTR
NBWinTypes -> #UINT
NBWinTypes -> #UINT32
NBWinTypes -> #UINT64
NBWinTypes -> #UINT_PTR
NBWinTypes -> #ULONG
NBWinTypes -> #ULONG32
NBWinTypes -> #ULONG64
NBWinTypes -> #ULONGLONG
NBWinTypes -> #ULONG_PTR
NBWinTypes -> #USHORT
NBWinTypes -> #USN
NBWinTypes -> #VOID
NBWinTypes -> #WCHAR
NBWinTypes -> #WNDCLASSEX
NBWinTypes -> #WNDPROC
NBWinTypes -> #WORD
NBWinTypes -> #WPARAM
NativeBoostConstants -> #ErrInvalidPlatformId
NativeBoostConstants -> #ErrInvalidPrimitiveVoltageUse
NativeBoostConstants -> #ErrNoNBPrimitive
NativeBoostConstants -> #ErrNoNativeCodeInMethod
NativeBoostConstants -> #ErrNotEnabled

NativeBoostConstants -> #ErrRunningViaInterpreter
NativeBoostConstants -> #Linux32PlatformId
NativeBoostConstants -> #Mac32PlatformId
NativeBoostConstants -> #NBErrorBase
NativeBoostConstants -> #NBErrorDescriptions
NativeBoostConstants -> #NBPrimErrBadArgument
NativeBoostConstants -> #NBPrimErrBadIndex
NativeBoostConstants -> #NBPrimErrBadMethod
NativeBoostConstants -> #NBPrimErrBadNumArgs
NativeBoostConstants -> #NBPrimErrBadReceiver
NativeBoostConstants -> #NBPrimErrGenericFailure
NativeBoostConstants -> #NBPrimErrInappropriate
NativeBoostConstants -> #NBPrimErrLimitExceeded
NativeBoostConstants -> #NBPrimErrNamedInternal
NativeBoostConstants -> #NBPrimErrNoCMemory
NativeBoostConstants -> #NBPrimErrNoMemory
NativeBoostConstants -> #NBPrimErrNoModification
NativeBoostConstants -> #NBPrimErrNotFound
NativeBoostConstants -> #NBPrimErrObjectMayMove
NativeBoostConstants -> #NBPrimErrUnsupported
NativeBoostConstants -> #NBPrimNoErr
NativeBoostConstants -> #Win32PlatformId
NetNameResolver -> #ResolverBusy
NetNameResolver -> #ResolverError
NetNameResolver -> #ResolverMutex
NetNameResolver -> #ResolverReady
NetNameResolver -> #ResolverUninitialized
OCASTTranslator -> #OptimizedMessages
PNGReadWriter -> #BPP
PNGReadWriter -> #BlockHeight
PNGReadWriter -> #BlockWidth
PNGReadWriter -> #StandardColors
PNGReadWriter -> #StandardSwizzleMaps
ParseNode -> #Bfp
ParseNode -> #BtpLong
ParseNode -> #CodeBases
ParseNode -> #CodeLimits
ParseNode -> #DblExtDoAll
ParseNode -> #Dup
ParseNode -> #EndMethod
ParseNode -> #EndRemote
ParseNode -> #Jmp
ParseNode -> #JmpLimit
ParseNode -> #JmpLong
ParseNode -> #LdFalse
ParseNode -> #LdInstLong
ParseNode -> #LdInstType
ParseNode -> #LdLitIndType
ParseNode -> #LdLitType
ParseNode -> #LdMinus1
ParseNode -> #LdNil
ParseNode -> #LdSelf
ParseNode -> #LdSuper
ParseNode -> #LdTempType
ParseNode -> #LdThisContext
ParseNode -> #LdTrue
ParseNode -> #LoadLong
ParseNode -> #LongLongDoAll
ParseNode -> #NodeFalse
ParseNode -> #NodeNil
ParseNode -> #NodeSelf
ParseNode -> #NodeSuper
ParseNode -> #NodeThisContext
ParseNode -> #NodeTrue
ParseNode -> #Pop
ParseNode -> #Send
ParseNode -> #SendLimit
ParseNode -> #SendLong
ParseNode -> #SendLong2
ParseNode -> #SendPlus
ParseNode -> #SendType
ParseNode -> #ShortStoP
ParseNode -> #StdLiterals
ParseNode -> #StdSelectors
ParseNode -> #StdVariables
ParseNode -> #Store
ParseNode -> #StorePop
ProcessorScheduler -> #HighIOPriority
ProcessorScheduler -> #LowIOPriority
ProcessorScheduler -> #SystemBackgroundPriority
ProcessorScheduler -> #SystemRockBottomPriority
ProcessorScheduler -> #TimingPriority
ProcessorScheduler -> #UserBackgroundPriority
ProcessorScheduler -> #UserInterruptPriority
ProcessorScheduler -> #UserSchedulingPriority
RBAbstractClass -> #LookupSuperclass
RBClass -> #LookupComment
RBScanner -> #PatternVariableCharacter
RBScanner -> #PatternVariableCharacter
RBScanner -> #classificationTable
RBTransformationRule -> #RecursiveSelfRule
RealEstateAgent -> #StaggerOffset
RemotesManager -> #addRemoteIcon
RemotesManager -> #editRemoteIcon
RemotesManager -> #removeRemoteIcon
RxMatcher -> #Cr
RxMatcher -> #Lf
RxParser -> #BackslashConstants
RxParser -> #BackslashSpecials
RxsPredicate -> #EscapedLetterSelectors
RxsPredicate -> #NamedClassSelectors
SHA1 -> #K1
SHA1 -> #K2
SHA1 -> #K3
SHA1 -> #K4
Scanner -> #DoItCharacter
SetElement -> #NilElement
Socket -> #Connected
Socket -> #DeadServer
Socket -> #InvalidSocket
Socket -> #OtherEndClosed
Socket -> #TCPSocketType
Socket -> #ThisEndClosed
Socket -> #UDPSocketType
Socket -> #Unconnected
Socket -> #WaitingForConnection
String -> #AsciiOrder
String -> #CSLineEnders
String -> #CSNonSeparators

String -> #CSSeparators
String -> #CaseInsensitiveOrder
String -> #CaseSensitiveOrder
String -> #CrLfExchangeTable
String -> #LowercasingTable
String -> #Tokenish
String -> #TypeTable
String -> #UppercasingTable
TextConstants -> #BS
TextConstants -> #BS2
TextConstants -> #Basal
TextConstants -> #Bold
TextConstants -> #CR
TextConstants -> #Centered
TextConstants -> #Clear
TextConstants -> #CrossedX
TextConstants -> #CtrlA
TextConstants -> #CtrlB
TextConstants -> #CtrlC
TextConstants -> #CtrlD
TextConstants -> #CtrlDigits
TextConstants -> #CtrlE
TextConstants -> #CtrlF
TextConstants -> #CtrlG
TextConstants -> #CtrlH
TextConstants -> #CtrlI
TextConstants -> #CtrlJ
TextConstants -> #CtrlK
TextConstants -> #CtrlL
TextConstants -> #CtrlM
TextConstants -> #CtrlN
TextConstants -> #CtrlO
TextConstants -> #CtrlOpenBrackets
TextConstants -> #CtrlP
TextConstants -> #CtrlQ
TextConstants -> #CtrlR
TextConstants -> #CtrlS
TextConstants -> #CtrlT
TextConstants -> #CtrlU
TextConstants -> #CtrlV
TextConstants -> #CtrlW
TextConstants -> #CtrlX
TextConstants -> #CtrlY
TextConstants -> #CtrlZ
TextConstants -> #Ctrla
TextConstants -> #Ctrlb
TextConstants -> #Ctrlc
TextConstants -> #Ctrld
TextConstants -> #Ctrle
TextConstants -> #Ctrlf
TextConstants -> #Ctrlg
TextConstants -> #Ctrlh
TextConstants -> #Ctrli
TextConstants -> #Ctrlj
TextConstants -> #Ctrlk
TextConstants -> #Ctrll
TextConstants -> #Ctrlm
TextConstants -> #Ctrln
TextConstants -> #Ctrlo
TextConstants -> #Ctrlp
TextConstants -> #Ctrlq
TextConstants -> #Ctrlr
TextConstants -> #Ctrls
TextConstants -> #Ctrlt
TextConstants -> #Ctrlu
TextConstants -> #Ctrlv
TextConstants -> #Ctrlw
TextConstants -> #Ctrlx
TextConstants -> #Ctrly
TextConstants -> #Ctrlz
TextConstants -> #DefaultBaseline
TextConstants -> #DefaultFontFamilySize
TextConstants -> #DefaultLineGrid
TextConstants -> #DefaultMarginTabsArray
TextConstants -> #DefaultMask
TextConstants -> #DefaultRule
TextConstants -> #DefaultSpace
TextConstants -> #DefaultTab
TextConstants -> #DefaultTabsArray
TextConstants -> #ESC
TextConstants -> #EndOfRun
TextConstants -> #Enter
TextConstants -> #Italic
TextConstants -> #Justified
TextConstants -> #LeftFlush
TextConstants -> #LeftMarginTab
TextConstants -> #RightFlush
TextConstants -> #RightMarginTab
TextConstants -> #Space
TextConstants -> #Tab
TextConstants -> #TextSharedInformation
TextContainer -> #OuterMargin
TextConverter -> #latin1Encodings
TextConverter -> #latin1Map
ThumbnailMorph -> #EccentricityThreshhold
ThumbnailMorph -> #RecursionMax
Transcripter -> #Icon
TransferMorph -> #CopyPlusIcon
UCSTable -> #GB2312Table
UCSTable -> #JISX0208Table
UCSTable -> #KSX1001Table
UCSTable -> #Latin1Table
Unicode -> #Cc
Unicode -> #Cf
Unicode -> #Cn
Unicode -> #Co
Unicode -> #Cs
Unicode -> #DecimalProperty
Unicode -> #GeneralCategory
Unicode -> #Ll
Unicode -> #Lm
Unicode -> #Lo
Unicode -> #Lt
Unicode -> #Lu
Unicode -> #Mc
Unicode -> #Me
Unicode -> #Mn
Unicode -> #Nd
Unicode -> #Nl

Unicode -> #No
Unicode -> #Pc
Unicode -> #Pd
Unicode -> #Pe
Unicode -> #Pf
Unicode -> #Pi
Unicode -> #Po
Unicode -> #Ps
Unicode -> #Sc
Unicode -> #Sk
Unicode -> #Sm
Unicode -> #So
Unicode -> #ToCasefold
Unicode -> #ToLower
Unicode -> #ToUpper
Unicode -> #Zl
Unicode -> #Zp
Unicode -> #Zs
ZipConstants -> #BaseDistance
ZipConstants -> #BaseLength
ZipConstants -> #BitLengthOrder
ZipConstants -> #DistanceCodes
ZipConstants -> #DynamicBlock
ZipConstants -> #EndBlock
ZipConstants -> #ExtraBitLengthBits
ZipConstants -> #ExtraDistanceBits
ZipConstants -> #ExtraLengthBits
ZipConstants -> #FixedBlock
ZipConstants -> #FixedDistanceTree
ZipConstants -> #FixedLiteralTree
ZipConstants -> #HashBits
ZipConstants -> #HashMask
ZipConstants -> #HashShift
ZipConstants -> #MatchLengthCodes
ZipConstants -> #MaxBitLengthBits
ZipConstants -> #MaxBitLengthCodes
ZipConstants -> #MaxBits
ZipConstants -> #MaxDistCodes
ZipConstants -> #MaxDistance
ZipConstants -> #MaxLengthCodes
ZipConstants -> #MaxLiteralCodes
ZipConstants -> #MaxMatch
ZipConstants -> #MinMatch
ZipConstants -> #NumLiterals
ZipConstants -> #Repeat11To138
ZipConstants -> #Repeat3To10
ZipConstants -> #Repeat3To6
ZipConstants -> #StoredBlock
ZipConstants -> #WindowMask
ZipConstants -> #WindowSize
ZipFileConstants -> #CentralDirectoryFileHeaderSignature
ZipFileConstants -> #CompressionDeflated
ZipFileConstants -> #CompressionLevelDefault
ZipFileConstants -> #CompressionLevelNone
ZipFileConstants -> #CompressionStored
ZipFileConstants -> #DataDescriptorLength
ZipFileConstants -> #DefaultDirectoryPermissions
ZipFileConstants -> #DefaultFilePermissions
ZipFileConstants -> #DeflatingCompressionFast
ZipFileConstants -> #DeflatingCompressionMaximum
ZipFileConstants -> #DeflatingCompressionNormal
ZipFileConstants -> #DeflatingCompressionSuperFast
ZipFileConstants -> #DirectoryAttrib
ZipFileConstants -> #EndOfCentralDirectorySignature
ZipFileConstants -> #FaMsdos
ZipFileConstants -> #FaUnix
ZipFileConstants -> #FileAttrib
ZipFileConstants -> #IfaBinaryFile
ZipFileConstants -> #IfaTextFile
ZipFileConstants -> #LocalFileHeaderSignature
ZnBase64Encoder -> #DefaultAlphabet
ZnBase64Encoder -> #DefaultInverse
ZnByteEncoder -> #ByteTextConverters
ZnConstants -> #HTTPStatusCodes
ZnHeaders -> #CommonHeaders
ZnMimeType -> #ExtensionsMap
ZnUTF8Encoder -> #ByteASCIISet
ZnUTF8Encoder -> #ByteUTF8Encoding

Notes