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

```
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

```
def edge(clock source, clock target,
        int out, int initialTokens, int in) ≜
        clock read
        clock write
        source = (write ▼.(1.0^(out−1))^ω)
        ∧ 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
    |r w|
    r := self localClock: #read.
    w := self localClock: #write.

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

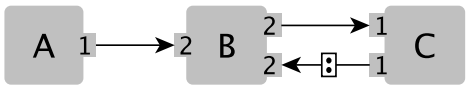


Figure 1: An example of an SDF graph

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	$a \prec b$	$a \prec b$	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)

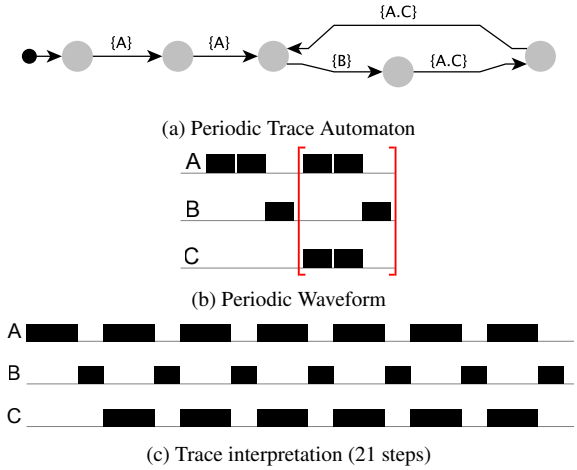
3.4 Syntactic Differences and Synonyms

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.

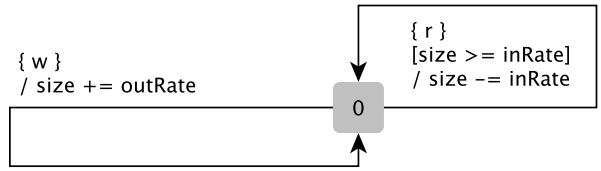


Figure 3: Automaton encoding the SDF execution policy

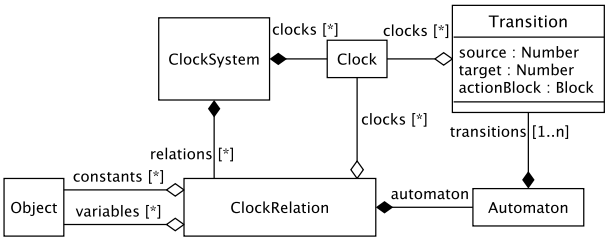


Figure 4: CLOCKSystem model (abstract syntax)

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.

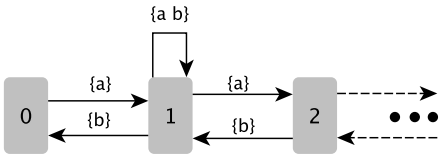


Figure 5: The infinite automaton of the $a \prec b$ relation

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

```
KernelLibrary>>#strictPrecedence
    ^ [ :s :a :b |
    "unbounded strict precedence"
    s = 0
        ifTrue:  [ {
            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 $a \prec b$ 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

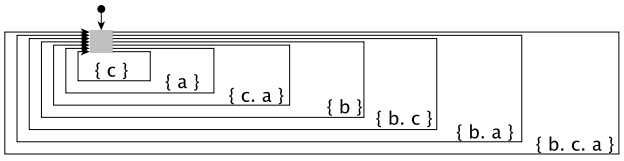


Figure 6: All possible behaviors of a specification with 3 independent clocks

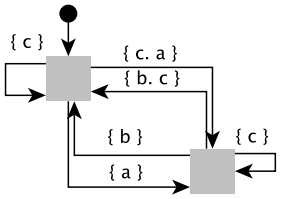


Figure 7: All possible behaviors of a specification with 3 clocks (a, b, c) where the ticks of a alternate with the ones of b.

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

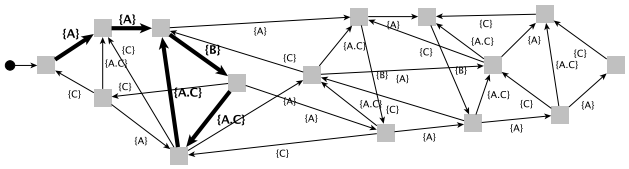


Figure 8: Exhaustive reachability analysis result for the SDF example with channel capacity bounded at 4 tokens.

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. .