smalltalkCI

A Continuous Integration Framework for Smalltalk Projects

Fabio Niephaus
Hasso Plattner Institute
University of Potsdam, Germany
fniephaus@acm.org

Dale Henrichs
GemTalk Systems
dale.henrichs@gemtalksystems.com

Marcel Taeumel
Hasso Plattner Institute
University of Potsdam, Germany
marcel.taeumel@hpi.de

Tobias Pape
Hasso Plattner Institute
University of Potsdam, Germany
tobias.pape@hpi.de

Tim Felgentreff
Hasso Plattner Institute
University of Potsdam, Germany
tim.felgentreff@hpi.de

Robert Hirschfeld
Hasso Plattner Institute
University of Potsdam, Germany
robert.hirschfeld@hpi.de

Abstract

Continuous integration (CI) is a programming practice that reduces the risk of project failure by integrating code changes multiple times a day. This has always been important to the Smalltalk community, so custom integration infrastructures are operated that allow CI testing for Smalltalk projects shared in Monticello repositories or traditional changesets.

In the last few years, the open hosting platform GitHub has become more and more popular for Smalltalk projects. Unfortunately, there was no convenient way to enable CI testing for those projects.

We present smalltalkCI, a continuous integration framework for Smalltalk. It aims to provide a uniform way to load and test Smalltalk projects written in different Smalltalk dialects. smalltalkCI runs on Linux, macOS, and on Windows and can be used locally as well as on a remote server. In addition, it is compatible with Travis CI and AppVeyor, which allows developers to easily set up free CI testing for their GitHub projects without having to run a custom integration infrastructure.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—Testing tools, Diagnostics

Keywords Smalltalk, Continuous Integration, Travis CI, AppVeyor, Coverage Testing

[Copyright notice will appear here once 'preprint' option is removed.]

1. Introduction

Working collaboratively with a shared code base is a cognitive challenge. Adapting best practices such as test-driven development increases code quality [9] and therefore its readability and maintainability. In order to make a project usable for others, it is often necessary to use dependency management tools. However, this still does not guarantee that a given program works for another developer or even for an end-user. Continuous integration (CI) [5] is an extreme programming practice that allows developers to change and refactor code while reducing the risk of project failure [2, 8].

The Smalltalk community has always been interested in sharing code. For example, one of the first features of Smalltalk-80 is a file out mechanism [12]. Later, Monticello repositories have become the de facto standard format to share Smalltalk code in different Smalltalk dialects for years. With the success of common version control systems (VCS) such as Git and open-source hosting platforms like GitHub and Bitbucket, Smalltalk code is more and more shared on these platforms in a Git-compatible format called FileTree[1].

With SUnit, Smalltalkers have built the first unit-testing framework [1] to enable test-driven development which has been essential for Smalltalk code quality ever since [16]. This stresses how important testing is for the Smalltalk community. A common way to enable CI testing for Smalltalk projects hosted on platforms such as SqueakSource[2] or SmalltalkHub[3] is to run a custom integration infrastructure, for example a Jenkins[4] instance.

In this paper, we present smalltalkCI, a continuous integration framework for Smalltalk projects. It is the successor to

[1] https://github.com/dalehenrich/filetree
[2] http://squeaksource.com/
[3] http://smalltalkhub.com/
[4] https://jenkins.io/

the builderCI[5] framework and aims to be reliable, lightweight, and easy to use. Since it is deeply integrated with Travis CI[6] and compatible to AppVeyor[7], both are continuous integration services for GitHub projects, it enables developers to easily set up free, continuous testing for Smalltalk projects on GitHub. In addition, it runs on Linux, macOS, and Windows, is compatible with different Smalltalk dialects such as Squeak, Pharo, and GemStone, and operates in a uniform way. This allows CI testing of cross-dialect projects such as Seaside [7], Fuel [6], Metacello, Parasol, and many more.

We believe that the Smalltalk community would benefit from shifting towards open platforms in a similar way other communities around programming languages are benefiting. This includes better visibility of the work that the community is doing as well as less effort that needs to be put into mostly administrative tasks such as running a CI server. We built smalltalkCI to make it easier to use these services and to encourage developers to share their Smalltalk projects on common, and more popular hosting platforms.

This paper explains how we designed and implemented a framework that

- supports CI testing for Smalltalk projects in a uniform way and with low effort;
- supports different Smalltalk dialects and operating systems;
- supports developers to debug their code locally and remotely; and
- is a mature substitute for a custom CI infrastructure that comes for free when hosting projects on GitHub.

In the following section, we introduce related tools, design decisions, and concepts. Then in sections 3 and 4, we explain how smalltalkCI is implemented, how it can be used, and how it is integrated into Travis CI. In section 5, we discuss ideas on how smalltalkCI can be used in the future. Finally, we conclude in section 6.

2. Background

Traditional Smalltalk environments provide the ability to share code in the FileOut code format which is a text-based format that simply holds compiler expressions. Multiple changes or packages can be exported using this format in so called changesets. However, the code format does not support dependency management in any way, but more importantly, it is inconvenient to manage code like this with modern VCSs. Not only can this code format be difficult to parse for human readers because fileouts can be quite long and unstructured. This already makes it hard to merge code if necessary. Additionally, the code format contains carriage return and other control characters which cause many merge conflicts in the first place. The FileTree format attempts to solve these problems by splitting multiple packages into separate directories and by exporting methods into separate files. This makes it much more convenient to manage Smalltalk code with VCSs such as Git or Subversion. It is also compatible with the Metacello package management system, which provides the ability to declare dependencies within a project. Therefore, these two technologies have made it possible to host Smalltalk projects on common platforms such as GitHub.

The need for CI support for these kinds of projects has resulted in the development of builderCI. This framework allows to build and test Smalltalk projects hosted on GitHub with Travis CI. However, it requires non-trivial boilerplate code to be added to a project and it uses a few tricks to be able to run on Travis CI. It also is difficult to debug build problems because it is not convenient and safe to run the framework locally and sometimes problems are caused by builderCI itself. Nonetheless, it is great that this project added CI support for Smalltalk projects that are not being hosted on Smalltalk-only platforms which ultimately motivated us to build smalltalkCI.

Travis CI offers free CI services with support for Linux and macOS for public GitHub repositories. For this, so called builds are performed whenever new code has been pushed to a repository. A build consists of one or more jobs and for each job, a worker machine is scheduled to build and test a project. How a project is built and tested needs to be specified in a `.travis.yml` file which has to live in the repository's root directory. In this file, it is possible to customize any step of the build process. It also holds the build matrix which specifies which jobs Travis CI should run for each build. AppVeyor is a comparable service which supports Windows builds and which can be set up in a similar fashion.

Current variants of the Smalltalk-80 programming environment [13] include Squeak [14], Pharo [4], and GemStone [15]. All of them come in different versions including bleeding edge or trunk versions. One motivation for setting up CI testing for a Smalltalk project is that a project can be automatically tested in different versions of the same Smalltalk dialect or in images of other Smalltalk dialects. This supports developers to maintain a project using their favorite Smalltalk image while making sure that the project works in all supported versions and dialects.

2.1 Design Decisions

For continuous integration, "it is important to have tools that support a fast integration/build/test cycle" [2]. We wanted that the time for an integration test is no longer than five minutes in average. Therefore, we aimed for a lightweight system with only few dependencies. A build should fail as early as possible in order to provide fast feedback in case of build problems and to avoid stuck build processes. Further, whenever adding a feature, we evaluate how it is impacting the time to run and we then consider if it is worth the trade-off or if there is a more time efficient alternative.

[5] https://github.com/dalehenrich/builderCI

[6] https://travis-ci.org/

[7] https://www.appveyor.com/

Moreover, we did not want smalltalkCI to have dependencies that need to be loaded externally in order to make it as reliable and robust as possible. Otherwise builds could fail if an external dependency became inaccessible. Since many projects already have quite a few external dependencies, we did not want smalltalkCI to add to the problem. In addition, we also wanted to have its entire code base including the Bash scripts and the Smalltalk code as well as its dependencies entirely in one repository. For this reason, hosting smalltalkCI's Smalltalk code in for example a Monticello repository was out of the question for us.

We initially started to export smalltalkCI's code in a single Smalltalk file that can be simply filed into an image. But we quickly realized that this approach is very inconvenient because it caused many Git merge conflicts which made maintaining the code base unnecessarily difficult. Instead, we then agreed to export the code in the FileTree format. This implies that in order to load smalltalkCI, Metacello and FileTree need to be installed in an image. Both are required to load most Smalltalk projects, especially the ones hosted on GitHub, anyway.

Pharo and GemStone images come with Metacello and FileTree pre-installed and hence can load smalltalkCI and projects without further ado. In Squeak, smalltalkCI can easily install Metacello[8] and it takes about a minute to load both, Metacello and FileTree, before being able to perform an actual integration test. This adds additional overhead which is especially noticeable when building smaller projects, whose tests only need several seconds to run. For this reason, smalltalkCI uses prepared Squeak images which have Metacello and FileTree pre-configured. This also makes them more robust because less external code needs to be loaded. Such pre-configured images, however, have to be maintained and checked for compatibility problems and are therefore only created for stable releases of Squeak.

The bootstrapping process is written in Bash and we tried to keep this code small and simple. The bootstrapping code currently accounts for approximately 38 % of smalltalkCI's code base[9]. We thought that everything that can be done in the Smalltalk image should happen in the image.

In addition, we tried to keep the total file size of smalltalkCI as small as possible. Every additional file that we add to the repository needs to be downloaded by a CI infrastructure, no matter whether it is Travis CI or a Jenkins instance. Currently, the entire code of smalltalkCI can be compressed into a 0.5 MB zip file. Adding for example an entire Smalltalk helper image with 30 MB in size would have increased the traffic to download the framework 1,000 times from 500 MB to 30.5 GB. For this reason, we decided against using a Smalltalk image that performs the integration test. Instead, we opted for a light Bash script layer while making sure that we only add files that are required for a build.

[8] https://github.com/dalehenrich/metacello-work

[9] According to https://github.com/hpi-swa/smalltalkCI.

2.2 The Configuration Concept

For an integration test, it is necessary to somehow specify how a project can be loaded and tested. Initially, we thought it would be a good idea to just add smalltalkCI-specific configuration information to the `.travis.yml` which Travis CI expects. This would have made smalltalkCI dependent on Travis CI which we wanted to avoid. We would have also needed to come up with similar solutions for other platforms like AppVeyor. Instead, we opted for a dedicated configuration file following a common pattern used in similar tools for other programming languages such as a `Gemfile` in Ruby, a `requirements.txt` in Python, or a `package.json` in NodeJS.

By default, the configuration file is expected to be called `.smalltalk.ston` or `smalltalk.ston`. In case one wants to use a different name, a single line needs to be added to the Travis CI configuration or if run manually, the custom file needs to be provided explicitly as a parameter. This makes common use cases very simple while allowing more advanced use cases with minimal additional effort.

For the configuration file format, we chose to use the Smalltalk Object Notation (STON)[10]. This way it is possible to specify Smalltalk objects in a JSON-like file which can then be added to the repository.

Currently, the configuration file only serves the purpose of telling smalltalkCI how to run integration tests for a given projects. But one could also use it as a cross-platform load specification, respectively making every repository that uses smalltalkCI instantly loadable in a consistent way for users.

3. Implementation

smalltalkCI is able to perform a full integration test for a given Smalltalk project. For this, it needs to download the Smalltalk target image and a corresponding virtual machine. Then it needs to initiate the integration process inside the target image which loads the project of interest and executes its test suite. Finally, smalltalkCI reports the test results it collected.

The initial version of smalltalkCI consisted of a small Bash script and a short Smalltalk script and was able to perform a CI test in a single Smalltalk dialect. We then used different abstraction layers in order to make the framework compatible to other dialects and to make it easily extendable while keeping its list of dependencies as short as possible. In addition, we incorporated proven concepts from builderCI and thought about ways to eliminate its shortcomings. smalltalkCI is tested by itself on Travis CI and AppVeyor with tests that cover its most important components and various cases. This helps to ensure that updates, which usually roll out automatically, do not break builds for everyone.

[10] https://github.com/svenvc/ston/blob/master/ston-paper.md

3.1 Bootstrapping

The bootstrapping of an integration test is done in a set of Bash scripts. They prepare the build and the caching directories, locate the build configuration, and invoke a dialect-specific build process. For each supported Smalltalk dialect, there is a dedicated script which knows where to download a Smalltalk image and an appropriate virtual machine. These files are first downloaded to the caching directory and then extracted to the build directory. This way, downloaded files do not need to be downloaded twice on the same system when running another integration test for the same platform. This is especially useful for debugging problems on a local machine. Once all files are in place, the corresponding smalltalkCI packages are loaded into the image. Finally, the bootstrapping process completes by first loading the project with

```
SmalltalkCI load: '/path/to/smalltalk.ston'
```

and then by initiating the integration test with

```
SmalltalkCI test: '/path/to/smalltalk.ston'.
```

3.2 Loading and Testing a Project

smalltalkCI's only external dependency is the STON package which is already available in Pharo and GemStone. In Squeak, the package is loaded directly from smalltalkCI's repository alongside with smalltalkCI. This dependency is needed to be able to load a configuration object from the configuration file provided by the user. The main object smalltalkCI expects is called a `SmalltalkCISpec`. Listing 1 shows the contents of a minimal configuration file which specifies such a specification.

Listing 1. Minimal `SmalltalkCISpec` example.

```
SmalltalkCISpec {
  #loading : [
    SCIMetacelloLoadSpec {
      #baseline : 'MyProject',
      #directory : 'packages',
      #platforms : [ #squeak, #pharo, #gemstone ]
    }
  ]
}
```

A `SmalltalkCISpec` has an instance variable `loading` which is expected to be a list of load specifications. A load specification inherits from `SCIAbstractLoadSpec` and describes how a project is loaded. For this, a load specification has an instance variable `platforms` which specifies to which Smalltalk dialects it is compatible. This way, one can use different load specifications for different dialects while making the common use case as simple as possible.

At the moment, smalltalkCI provides the classes `SCIMetacelloLoadSpec`, `SCIMonticelloLoadSpec`, and `SCIGoferLoadSpec` which allow to load projects via Metacello[11], Monticello, and Gofer[12] respectively. The above example instructs Metacello to load the Metacello baseline `BaselineOfMyProject`, which can be found in subdirectory called `packages/` relative to the project's root directory, in Squeak, Pharo, and GemStone.

Listing 2. `SmalltalkCI»load` (simplified).

```
load
[ self prepareForLoading.
  "Install all specs for the current platform"
  self compatibleLoadSpecs
    do: [ :each | each loadProjectOn: self ]
] ensure: [ self finishUpAfterLoading ]
```

Listing 2 shows the main entry point for loading a project after a configuration has been processed by the STON parser. Everything that needs to happen in order to set up an image for loading a project is done in `SmalltalkCI»prepareForLoading` which is specialized for each dialect. For example in Squeak, we need to load a custom tool set (`SCISqueakToolSet`) which quits the image as soon as an error is raised in headless mode. We also temporarily replace the default `Transcript` with an instance of `SCISqueakTranscript` which redirects the transcript to `stdout`.

The method `SmalltalkCI»finishUpAfterLoading` is its counterpart and basically reverts all changes in order to bring the image in its initial state.

Once the test image is prepared, smalltalkCI loads all compatible load specifications and then saves and closes the image.

Listing 3. `SmalltalkCI»test` (simplified).

```
test
| success |
self prepareForTesting.
self isCoverageTestingEnabled
  ifTrue: [ success := self runTestsWithCoverage ]
  ifFalse: [ success := self runTests ].
SmalltalkCI isHeadless ifFalse: [ self halt ].
success
  ifTrue: [
    SmalltalkCI closeWithExitCode: 0 ]
  ifFalse: [
    SmalltalkCI saveImage.
    SmalltalkCI closeImageWithExitCode: 1 ]
```

The entry point which initiates the integration test is shown in listing 3. Similar to `SmalltalkCI»prepareForLoading`, the image is prepared for testing first. Then all tests are run according to the configuration and, if requested, with coverage reporting enabled. We will talk about code coverage in detail in section 4.6. The implementation of `SmalltalkCI»runTests` is platform-specific. In order to be able to print test results

[11] https://github.com/dalehenrich/metacello-work

[12] http://www.lukas-renggli.ch/blog/gofer

in a uniform way, smalltalkCI exports the test results in the JUnit XML format. In Pharo, it uses Hudson's `HDTestReport` to build and run a test suite. This report already exports the results in the expected format. For Squeak and GemStone, we ported `HDTestReport` in order to make it compatible with each platform. In case the headfull mode is enabled, the image will halt before the image is programmatically closed. This allows to inspect the image right after the integration testing phase. Finally and if all tests passed, the image will be closed without saving, effectively reverting the image back to the point when it was saved right after loading the project. This ensures that the image is free of any side-effects that poorly written tests may have caused during their execution. In case of test failures or errors, the image is saved to persist any defects and then closed with a non-zero exit code.

Once `SmalltalkCI»test` finished, the integration process is complete and the workflow finishes on Bash level. The corresponding build directory is searched for test results in the XML format which are then printed in a uniform way to `stdout`.

```
311 ##############################################################
312 # SmalltalkCI: 6 Tests, 3 Failures, 1 Errors in 0.006s
313 ##############################################################
314
315 SmalltalkCI.Example.Tests
316   ✗ testAssertError (0.002 seconds)
323   ✗ testError (0.002 seconds)
329   ✗ testFailure (0.000 seconds)
336   ✓ testShouldFail (0.000 seconds)
337   ✓ testShouldPass (0.000 seconds)
338   ✗ testShouldPassUnexpectedly (0.000 seconds)
340
341     Executed 6 tests, with 3 failures and 1 errors in 0.006 seconds.
```

Figure 1. Example smalltalkCI build results in Travis CI log.

When running on Travis CI, some syntactic sugar is added to the output format which allows to fold lines in the Travis log (see figure 1). Clicking on one of the carets on the left will display the error message and the corresponding stack trace. Then the script also checks if coverage results are available in the build directory and uploads them accordingly.

Finally, smalltalkCI exists with exit code `0` if the integration test was successful, otherwise it exists with a different exit code which indicates a failed build.

3.3 Running smalltalkCI

Although smalltalkCI was initially developed to run on Travis CI, it can easily be run locally, on AppVeyor, or on any other CI infrastructure. Travis CI's container-based infrastructure does not allow the use of `sudo` for security reasons which was one motivation not to use it in smalltalkCI. Besides, smalltalkCI only operates within its own directory. Therefore it is safe to run the framework on a local machine.

It is only necessary to download or clone smalltalkCI and to install all dependencies on the operating system level for the virtual machines that will be used. Then a build can be initiated by executing the `run.sh` script which looks for a smalltalkCI configuration file in the current working directory. This way it is possible to run integration tests by executing the script within a local Git repository. A custom configuration file can be provided as command line argument and then smalltalkCI will try to locate the project automatically. Besides, it supports different command line options that can be listed by calling `run.sh --help`.

By default a build process runs in headless mode and fails as soon as an error occurs. For debugging build problems locally, it is especially useful to enable headfull mode via `--headfull`. In this mode, one can see what is happening inside the image and usually a debugger window will open when a problem occurs. Additionally, the image will be kept open, so that one can rerun the tests using the `TestRunner` or further inspect the image after the integration test happened. In order to make debugging problems even easier, smalltalkCI prints instructions to reproduce a build locally at the end of a Travis CI or AppVeyor log if a build has failed.

4. Integration into Travis CI

Travis CI is a popular continuous integration service for projects hosted on GitHub. It is free for open source projects and supports a variety of programming languages and build environments. We have integrated smalltalkCI into the service in order to make it easy to test Smalltalk projects on Linux and macOS.

4.1 Smalltalk as Community-Supported Language

Travis CI provides the ability to add so called community-supported languages[13]. For adding Smalltalk, we had to implement a Ruby class which is part of Travis CI's code base. This class inherits from a generic `Script` class and overrides methods for the different build stages.

The first stage is called `configure` and allows the use of `sudo`. Therefore, all build environments that Travis CI supports can be prepared during this stage, especially the container-based environment which does not allow users to call `sudo` for security reasons. For Squeak and Pharo builds, all dependencies for the virtual machine that will be used for the build are installed during this stage. GemStone builds require additional work, for example configuring shared memory, preparing the host file, and starting GemStone's `netldi` service. The second stage is named `export` and simply exports environment variables from the `.travis.yml` which smalltalkCI will then detect and use for the build process. During the `setup` stage, a Travis CI worker downloads and extracts the latest version of smalltalkCI and then initializes its environment variables. Lastly, smalltalkCI is started by executing its `run.sh` script without arguments in the `script` stage by default. This stage can be overridden in the `.travis.yml` in order to, for example, run smalltalkCI in a debug or verbose mode.

[13] https://docs.travis-ci.com/user/languages/community-supported-languages/

4.2 Testing Common Smalltalk Projects with Travis CI

To enable CI testing with Travis CI, one needs to enable the service for the GitHub project of interest and then add a `.travis.yml`, which is a Travis CI-specific configuration file in the YAML format, to the repository.

Listing 4. Minimal `.travis.yml` example.

```
language: smalltalk
sudo: false

os:
  - linux
  - osx

smalltalk:
  - Squeak-5.0
  - Pharo-5.0
  - GemStone-3.3.0
```

Listing 4 shows an example configuration file for a Smalltalk project. The `language` key is set to `smalltalk` and instructs Travis CI to build the project with smalltalkCI. `sudo: false` makes sure that Linux builds are routed through the container-based infrastructure. This infrastructure does not support the use of `sudo`, but it is able to boot a worker within one to six seconds[14] which makes it the fastest build environment that Travis CI provides. The `os` key is used to enable testing on multiple operating systems. At the time of writing, Linux and macOS are supported by Travis CI. For Windows builds, it is possible to use AppVeyor in a similar way described in smalltalkCI's `README.md`. Lastly, a list of target Smalltalk images is provided via the `smalltalk` key. The list can consist of any two or more Smalltalk images that are supported by smalltalkCI (a list of all currently supported images can be found in smalltalkCI's `README.md`). In case the project should only be built in one image, a simple key-value pair can also be provided. Whenever commits are pushed to GitHub in this example, Travis would schedule six builds in total, trying to build and test the project in a Squeak, a Pharo, and a GemStone image on each, Linux and macOS.

For more complex CI setups, Travis CI provides the concept of a build matrix which can be used to add and remove specific jobs after the matrix is expanded. Additionally, it is possible to use a specific or multiple smalltalkCI configurations by using the `smalltalk_config` key to specify configuration file names which will also expand the build matrix accordingly.

4.3 Testing a GemStone Setup

Since GemStone is an object-oriented database based on Smalltalk [15], it is possible to run continuous integration tests in a similar way to other Smalltalk dialects. Compared to Squeak and Pharo, it does require more work in order to prepare the operating system for running the GemStone server. Fortunately, there is a comprehensive open source development kit for GemStone called GsDevKit[15] which ships with an installation script. We have ported this installation script to Ruby and run the same steps during the `configure` stage described in section 4.1 for GemStone builds in order to set up the worker. Furthermore, smalltalkCI uses the GsDevKit during GemStone builds for creating a stone and clients as well as for initiating and synchronizing the integration test. Setting up CI testing for a GemStone server is as easy as setting it up for Squeak or Pharo.

Listing 5. Example `.travis.yml` for a GemStone setup.

```
language: smalltalk
sudo: false

smalltalk: GemStone-3.3.0

matrix:
  include:
    - env: GSCI_CLIENTS="Pharo-5.0"
```

Listing 5 shows a very basic `.travis.yml` which sets up two jobs on Travis CI. One job simply runs the integration test for GemStone-3.3.0, the other one does the same but additionally runs a client-side integration test in a Pharo-5.0 client image. For additional clients, one can set the environment variable `GSCI_CLIENTS` to a list of different clients separated by spaces.

Besides, it is also possible to provide additional configuration information through a `SCIGemStoneServerConfigSpec` which is used to configure the stone. This spec has to be part of the project's `SmalltalkCISpec` and can be used to specify for example the default session name, paths to stone, and session configuration files, or the timezone used in the stone.

smalltalkCI is able to perform integration tests accordingly and delegates subtasks to the GsDevKit when necessary.

4.4 Caching

Travis CI is able to cache dependencies and directories in order to speed up builds. The directory caching mechanism simply checks a given directory for changes after a build and then zips and uploads it to Amazon S3. In the beginning of the next build, Travis CI will download and extract all cached directories, so that they are available in the same location.

When integrating and testing a Smalltalk project, it is necessary to download an image and a virtual machine which is unavoidable. Loading a project often takes longer than running its tests. But it would not be beneficial to cache project code and unfortunately, there currently is no way to cache external dependencies in a reasonable fashion or to upgrade loaded Smalltalk projects efficiently.

Also, builds on Travis CI would not benefit from caching the image and the virtual machine with the built-in caching mechanism because both have to be downloaded anyway, regardless of whether from the original source or from the Amazon S3 cache.

[14] https://docs.travis-ci.com/user/ci-environment

[15] https://github.com/GsDevKit/GsDevKit_home

But caching can be used to speed up GemStone builds. Preparing a stone is quite time consuming and the same stone file can be reused without a problem once it has been generated for a specific version. This allowed us to bring build times of five to twelve minutes for smalltalkCI itself down to only two to three minutes.

4.5 Travis CI for Non-GitHub Projects

smalltalkCI works well with Travis CI and with Smalltalk projects that are being exported using FileTree and are hosted on GitHub. However, it is possible to leverage Travis CI's free service for projects that are not necessarily hosted on GitHub. This is possible because smalltalkCI can load any project that Monticello can load. Unfortunately, it is still necessary to set up a GitHub repository to be able to enable Travis CI. But this repository only needs to hold a configuration file for each, Travis CI and smalltalkCI. Once everything is set up, a new CI build can be triggered by sending a simple HTTP request to Travis CI's API[16].

The Fuel team has found one way to automate this. The Pharo Jenkins server at ci.inria.fr polls for changes in the Fuel repository on SmalltalkHub using the URLTrigger plugin and then triggers a new build on Travis CI. Referring to a maintainer of Fuel, some reasons why Fuel is not being tested directly by the Jenkins server are that smalltalkCI is easier to set up, that it makes it simple to test Fuel in other dialects, and that it is much more reliable than the custom CI infrastructure.

SmalltalkHub already supports commit hooks which are HTTP requests that are sent to a designated URL after each push. However, those commit hooks currently only allow to perform HTTP `GET` requests. Once they are able to perform HTTP `POST` requests with custom headers, there would be no need to use a polling Jenkins job anymore because commit hooks can be used to directly trigger new builds on Travis CI instead.

4.6 Coverage Testing

When run on Travis CI, smalltalkCI also integrates with Coveralls[17], which is a tool for test coverage history and statistics. In order to enable test coverage reporting, one needs to enable Coveralls for the corresponding GitHub project and then needs to specify packages and classes via the project's smalltalkCI configuration for which the coverage should be determined. If coverage testing is enabled, smalltalkCI uses SUnit's `TestCoverage` which is also used for coverage testing in the original `TestRunner`.

Before running the test suite, all methods for all selected classes are being replaced by instances of `TestCoverage`. Each `TestCoverage` object can be called as method because it implements `run:with:in:`. Inside this method it calls the original method on the initial receiver after marking itself as visited. Once the test suite finished running all tests, all `TestCoverage` objects are checked if they have been visited and then uninstalled. This allows to determine if a method has been called or not. smalltalkCI then exports the coverage results in a JSON-like format which is then decorated with build information on Bash level and sent to the Coveralls service.

Finally, Coveralls analyzes the file, displays the coverage results, and links methods to their corresponding `.st` file in the GitHub repository if possible.

5. Future Work

smalltalkCI is already being used to build and test more than 90 projects on GitHub[18] and provides many features. Nonetheless, we are constantly extending its capabilities to make it even easier to use and to support more use cases.

In the near future, smalltalkCI will use a custom test runner which will give users more control over how tests are being executed. Also, we are planning to move the code that prints test results into the image to further decrease complexity of the Bash script layer.

We have recently added compatibility to AppVeyor which allows to run integration tests on Windows in a similar way tests run on Travis CI. But AppVeyor support is not yet complete and can be improved, for example coverage testing is currently not supported. Similarly, it would be interesting to set up smalltalkCI on other CI services like GitLab CI[19] or CircleCI[20].

Sharing Smalltalk code in Git repositories is still not as easy as it could be. First, the FileTree format saves meta data within the repository which for example contains the version history. This means that the version history is basically duplicated. It would be better to only use the Git commit history instead, especially because Git is unable to merge these meta data files automatically. This for example makes it impossible to merge pull requests without having to manually resolve conflicts. There are already ideas to remove the redundant meta data files which we want to support in the future. Moreover, the tools that are currently available to export Smalltalk code are still inconvenient to use. Exporting a new version for example requires to export the project with the Monticello Browser in the FileTree format and then it is still necessary to create a Git commit with the changed files. These tools also do not ship with Squeak yet, which is something we want to work on in the future as well. As the Smalltalk community is working on these format and tooling issues, we will update smalltalkCI accordingly.

Furthermore, we are planning to add support for other virtual machines. Currently, smalltalkCI selects the default virtual machine for an image automatically. In some use cases, it would be great to be able to use a different VM, e.g. an in-

[16] https://docs.travis-ci.com/user/triggering-builds

[17] https://coveralls.io

[18] According to https://git.io/v6lsh.

[19] https://about.gitlab.com/gitlab-ci/

[20] https://circleci.com/

terpreter VM or experimental VMs such as SqueakJS [11] or RSqueak/VM [10]. Additionally, we will need to add support for 64-bit images and virtual machines soon.

With the current approach, code coverage can currently only be determine per method, but it would be more accurate to determine the coverage per line. Hapao [3] is a test coverage tool which supports per-line coverage testing. We still need to evaluate if and how we can use Hapao in smalltalkCI.

Further, we are looking into tooling support for smalltalkCI configurations. It would be nice to have a user interface that can be used to create and manage configurations as well as a tool that can run multiple integration tests in different dialects from within an image. This would allow developers to check if their code runs in other versions of the same Smalltalk dialect or even in images of different dialects which would be especially useful for developing cross-dialect-compatible projects. The AutoTDD[21] project for instance already supports to run a test suite whenever related code changes. It also is able to parse tests results in the JUnit XML format which already makes it possible to view remote smalltalkCI test results inside an image.

As CI services are improving and extending their deployment capabilities, we are also planning to provide the ability to export the images to a hosting platform after the projects has been built successfully. smalltalkCI could for example prepare an all-in-one bundle which can then be uploaded as part of a GitHub release[22]. It might also be possible to use smalltalkCI in a similar way to build and release for example new Squeak-trunk images.

6. Conclusion

In this paper, we presented smalltalkCI, a continuous integration framework for Smalltalk projects. We explained how we strive to make it reliable, efficient, and easy to use, how it is implemented so that it supports different Smalltalk dialects, and how it can be used in various use cases.

With smalltalkCI, we were able to add Smalltalk support for the continuous integration services Travis CI and AppVeyor which now provide free CI testing for Smalltalk projects that are hosted on GitHub. Although the first version of smalltalkCI was released in the end of 2015, it is already used to build and test more than 90 projects on GitHub including projects like Seaside, Fuel, and Parasol. Also, more than ten different developers have contributed in more than 1,000 commits so far.

We hope that smalltalkCI continues to encourage Smalltalkers to share their projects on open platforms like GitHub which makes them more visible and allows better collaboration compared to Smalltalk-only platforms. Additionally, we hope that smalltalkCI supports Smalltalk developers in working on more cross-dialect projects from which the entire community benefits.

[21] https://github.com/HPI-SWA-Teaching/AutoTDD

[22] https://help.github.com/articles/about-releases/

Acknowledgments

We would like to thank all contributors to smalltalkCI from different Smalltalk communities for their help. A special thanks to Jonas Chromik, Steffen Kötte, Christopher Weyand, and Lennard Wolf from the bachelor project HPI-BP2015H as well as to Hiro Asari, Konstantin Haase, Sven Fuchs, Josh Kalderimis, Mathias Meyer, and the Travis CI team for helping us adding Smalltalk support with this framework to Travis CI. Finally, we gratefully acknowledge the financial support of HPI's Research School[23] and the Hasso Plattner Design Thinking Research Program[24].

References

[1] K. Beck. Simple smalltalk testing: With patterns. *The Smalltalk Report*, 4(2):16–18, 1994.

[2] K. Beck. *Extreme Programming Explained: Embrace Change*. An Alan R. Apt Book Series. Addison-Wesley, 2000.

[3] A. Bergel and V. Peña. Increasing test coverage with hapao. *Science of Computer Programming*, 79:86–100, 2014.

[4] A. P. Black, O. Nierstrasz, S. Ducasse, and D. Pollet. *Pharo by Example*. Square Bracket Associates, 2010.

[5] G. Booch. *Object Oriented Design: With Applications*. The Benjamin/Cummings Series in Ada and Software Engineering. Benjamin/Cummings Pub., 1991.

[6] M. Dias, M. M. Peck, S. Ducasse, and G. Arévalo. Fuel: A fast general purpose object graph serializer. *Software: Practice and Experience*, 44(4):433–453, 2014.

[7] S. Ducasse, A. Lienhard, and L. Renggli. Seaside: A flexible environment for building dynamic web applications. *Software, IEEE*, 24(5):56–63, 2007.

[8] P. M. Duvall, S. Matyas, and A. Glover. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[9] H. Erdogmus, M. Morisio, and M. Torchiano. On the effectiveness of the test-first approach to programming. *IEEE Transactions on Software Engineering*, 31(3):226–237, March 2005.

[10] T. Felgentreff, T. Pape, L. Wassermann, R. Hirschfeld, and C. F. Bolz. Towards reducing the need for algorithmic primitives in dynamic language vms through a tracing jit. In *Proceedings of the Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, ICOOOLPS'15, New York, NY, USA, 2015. ACM.

[11] B. Freudenberg, D. H. Ingalls, T. Felgentreff, T. Pape, and R. Hirschfeld. Squeakjs: a modern and practical smalltalk that runs in any browser. In *ACM SIGPLAN Notices*, volume 50, pages 57–66. ACM, 2014.

[12] A. Goldberg. Smalltalk-80: the interactive programming environment. 1984.

[13] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

[23] https://hpi.de/en/research/research-school.html

[24] https://hpi.de/en/dtrp/

[14] 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. *SIGPLAN Not.*, 32(10):318–326, Oct. 1997.

[15] D. J. Penney and J. Stein. Class modification in the gemstone object-oriented dbms. *SIGPLAN Not.*, 22(12):111–117, Dec. 1987.

[16] A. Sharp. *Smalltalk by Example: The Developer's Guide*. McGraw-Hill, Inc., New York, NY, USA, 1st edition, 1996.