Automatic Job Generation for Compiler Testing
Testing of Generated Compiler

Ludek Dolihal

Department of Information Systems,
Faculty of Information Technology,
Brno University of Technology
Brno, Czech Republic
Email: idolihal@fit.vutbr.cz

Tomas Hruska

Department of Information Systems,
Faculty of Information Technology,
Centre of Excellence IT4Innovations,
Brno University of Technology
Brno, Czech Republic
Email: hruska@fit.vutbr.cz

Abstract—**Hand in hand with the design of the new core goes the need for thorough testing, which is highly automated. Tools for hardware/software codesign allow very fast design of the new core and generation of the complete tool-chain. The tool-chain that is used for the programming of the newly developed core and also descriptions of the core in various languages are generated automatically and it is the role of automatic testing to ensure that there is no regression. As the pace of the development is high also the techniques for the testing must be able to cover the testing in very short period of time. In this article, we will introduce the generator of jobs for the continuous integration server Jenkins. Through the job generation we reach the higher level of automation of the whole process of the core development and also speed up the process of testing.**

***Keywords*–Compiler testing; Continuous integration; Hardware/Software codesign; Test generation.**

I. INTRODUCTION

Each software product must be tested. In the article, we will address the testing of tools for hardware/software codesign [1]. Hardware/software codesign deals with the design of the new Application Specific Instruction-set Processors (ASIPs). Such kind of systems can be found in wide variety of devices such as network routers or printers.

The production of ASIPs is growing as the need for the small and low power cores that can be used for specific purposes is still bigger. For example Texas Instruments released 4 new cores in the last 6 months [2]. Hence, this area is extremely important. The development of today's ASIPs must be done in a very short period of time. To do so, it is common to use the tools for hardware/software codesign. Some Architecture Description Language (ADL) is usually in the core of such systems [3]. The development is done in a modern Integrated Development Environment (IDE) that allows the designer to generate all the necessary tools, such as compiler, assembler and simulator. In the same environment the user is able to perform any step needed for the development of the core, such as simulation or profiling.

Such kind of development environment shortens the development time significantly [4]. However, each piece of software contains errors, and environments for hardware/software codesign are not an exception. Some of the tools are more error prone than others. From our point of view, the Software Development Kit (SDK), and especially the compiler, are the most critical parts. Because in case we have error in the compiler, the compiled program does not have to work properly. If the compiler does not work correctly, it is crucial to discover the error in the shortest possible time. For this purpose we use a continuous integration server to run testing jobs.

The continuous integration server will be used for execution of jobs that will be automatically generated. We will introduce the generator of the jobs that will bring the higher level of automation and also speed up the process of testing.

The paper is structured as follows: Section II, gives the short overview of the Lissom project. In Section III, we explain the continuous integration process. Section IV discuss the related work. In Sections V and VI, we explain the generator of the testing jobs and achieved results. Finally, in Section VII, we present the conclusions.

II. LISSOM PROJECT

In this section, we will describe the Lissom research project [5], which creates background for the testing methods that are described in this article. The Lissom project started in 2004 and is located at the Brno University of Technology, Faculty of Information Technology, Czech Republic.

The Lissom project has two main areas of interest. The first one is the development of the Architecture Description Language (ADL) called CodAL, which serves for the ASIP description. The description of the language can be found in detail here [6].

The second scope of the project is the generation of the full tool-chain from the description in the ADL CodAL. The generated tool-chain contains a C compiler, assembler, linker, disassembler, two types of simulators (instruction and cycle accurate), the debugger and few more tools. As the language is designed for description of the ASIPs, the scale of processors that can be described, without making any modifications to the language, is large.

However, there are also other ways how to utilise such language. One of them is to use the language for description of architectures that already exist. Hence, it is possible to model in the CodAL language architectures such as MIPS [7], ARM [8], RISC-V [9] and many others. The generated tool-chain or just separate tools can be used as a replacement of existing tools in case they are not in good shape. This gives large possibilities in case the core is upgraded and new tool-chain is needed. For certain cores also, some of the tools might be missing and by designing the given architecture in ADL the missing tool can be easily generated.

All the tools are generated from the description in the CodAL language. In the beginning, the model in the CodAL language is validated and compiled. The result of the compilation is the XML representation of the model. The XML format was chosen intentionally as there are other tools that use this form and there is also large number of generators and parsers working over XML.

Once the XML is created there are two tools working over it. The first tool is the tool-chain generator, also called toolsgen. The second one is the semantics extractor or semextr.

The tool-chain generator produces tools, such as simulator, assembler, debugger and many others. The tools that are generated by the tool-chain generator consist of two types of files. Both types of files are compiled and linked together.

1) Files that are platform independent are the same for all architectures. Into this category falls user interfaces with parsers of the command line arguments, or in case of profiler the generation of the graphical output.
2) Automatically generated files that contain the platform dependent information. Into this category belong the instruction decoders in the simulators or assembler printer in the C compiler.

The second tool is the semantics extractor. The execution of the extractor is the prerequisite for the compiler generation. Moreover, there are other tools that use the outputs of the semantics extractor, such as Quick EMUlator (QEMU) or documentation generator and also decompiler that is described in the thesis [10].

The main role of the semantics extractor is to extract the assembler syntax, binary encoding and semantics of each instruction described in the model.

The development of the new core in the Electronic Design Automation (EDA) tool [11] can be done very swiftly. The experienced designer can create an instruction accurate model of a core in a few hours. Modification of a core can be created even faster. It is very simple to add some instructions and/or create larger register field for example. This process can give birth to the versions of the processors that can be optimized for speed, size of the code or power consumption.

All such variants of the core should be tested, so there is a need for simple generation of the testing infrastructure. Hence we need a generator of the jobs, that will perform the testing. We need to speed up the whole process and reduce the amount of manual work.

III. Continuous Integration

In this section we will describe the Continuous Integration (CI) and introduce the job format, which we will use in the further sections.

The main idea of continuous integration [12] is to avoid the integration problems in the later stages of the development. The developers are encouraged to merge with the main development line several times a day and execute the tests over the merged line. By this approach they are encouraged to keep an eye on the integration continually.

The technique was mentioned for the first time by Grady Booch [13], and was called Booch method. Later it was adopted by extreme programmers and resulted in performing an integration in once or more times a day.

Today, the continuous integration servers are used in every larger company. The most widespread CI server is called Jenkins [14]. Jenkins is an open source automation server that can provide not only continuous integration but also continuous deployment. It uses the system of plugins to enhance the basic functionality. Nowadays, there are plugins available all the Version Control Systems (VCS) as well as plugins for visualisation of pipelines etc.

The basic block of the Jenkins server is called a job. The main action for every job is the execution. The job has the data that are typically taken from the VCS and action that is usually execution of some script.

The jobs are stored at the Master server. Master server is the computer that keeps the installation of the Jenkins and all the jobs are kept here. In case of the single master installation. The job is represented by a file in the XML format that is stored in the given folder on the Master server. The format in the markup language is in our case a great advantage as there is a lot of generators of the XML and also there are other tools that can work with the description.

A. Jenkins job format

Jenkins supports several types of jobs. The basic ones are the freestyle project and the multiconfiguration project. The main difference between the two is the fact that multiconfiguration project can be executed on multiple machines. There are also special types of jobs that are tied to the various plugins. There is a maven job, external job or various views.

Below we listed the basic description of the multiconfiguration job, as it is the job, which we are the most interested in. Though we need to work with the other job types as well, the configuration of the basic kind of job will be suffitient for demonstration purposes now.

```
<?xml version='1.0' encoding='UTF-8'?>
<matrix-project plugin="matrix-project@1.4
    ">
  <actions/>
  <description></description>
  <keepDependencies>false</
      keepDependencies>
  <properties>
    <com.sonyericsson.rebuild.
        RebuildSettings
     plugin="rebuild@1.22">
      <autoRebuild>false</autoRebuild>
     </com.sonyericsson.rebuild.
        RebuildSettings>
    <hudson.model.
        ParametersDefinitionProperty/>
  </properties>
  <scm class="hudson.scm.NullSCM"/>
  <canRoam>true</canRoam>
  <disabled>false</disabled>
  <blockBuildWhenDownstreamBuilding>false
  </blockBuildWhenDownstreamBuilding>
  <blockBuildWhenUpstreamBuilding>false
  </blockBuildWhenUpstreamBuilding>
  <triggers/>
  <concurrentBuild>false </concurrentBuild>
  <axes>
    <hudson.matrix.LabelAxis>
      <name>label </name>
      <values>
        <string>CentOS−6.5−32</string>
      </values>
    </hudson.matrix.LabelAxis>
  </axes>
  <builders>
    <hudson.tasks.Shell>
      <command>echo  \$(pwd) </command>
    </hudson.tasks.Shell>
  </builders>
  <publishers/>
  <buildWrappers/>
  <executionStrategy  class="hudson.matrix.
DefaultMatrixExecutionStrategyImpl">
    <runSequentially>false </
        runSequentially>
  </executionStrategy>
</matrix−project>
```

On the second line, we can see that it is the matrix project, which means that it can deploy multiple axis, and one of them is the configuration of the nodes. For simplicity the job does not download any data from the VCS. Another important tag is the one called *axes*. This tells us that this job is built only on one node called CentOS-6.5-32. It is important to note that this job does not have parameters. If it had, the parameters would be visible in the top of the configuration.

There are also sections *builders* and *publishers*. Section *builders* says that there is the shell script executed, and only command it runs it the *echo $(pwd)*. The job has no results, hence, the part *publishers* is empty. The execution strategy is default. It is important to know, how the configuration of the job looks like as we will work with the representation in the later sections.

IV. Related Work

Let us have a look at the current development at the field of the job generation. We can distinguish two types of solutions. The are tools in Jenkins that were designed for this purpose and then there are several works that try to deal with the problem of job generation outside of the Jenkins environment.

First we will have a look at the solutions inside the Jenkins. One of them is the template plugin [15]. Via the template project plugin the user can set up an template project containing the settings the user want to share. Is is possible to set for example the VCS repositories that are common for the jobs or the script that should be executed and so on. Then it is possible to create inside the Jenkins another project from the created template. So the generation has to be performed manually by using the template several times. Hence, the possibilities of the automation are limited.

Other possibility provided by Jenkins server itself is the job generator plugin [16]. This plugin is based on template, which is the job itself and the parameters, which can be global or local. This plugin is very powerful in combination with other plugins such as plugin for conditional resolution. However, it has limitations in form of what types of jobs can be generated and it can not use time triggers. Moreover, it is very difficult to generate more complex jobs. The hierarchy and conditions can become very complex and the whole process is error prone. We also did not find a way how to set the desired nodes in the multiconfiguration project.

Now we will mention several approaches that try to deal with job generation outside the Jenkins environment. The interesting ideas are proposed in the article at Jenkins User Conference [17]. The article deals with the automation of testing in the area of robotics. The author uses combination of various Jenkins plugins for packaging and static analysis. Nevertheless, the process of build and testing is very complicated and hardly maintainable. The author proposes use of Domain Specific Language (DSL) for specification of the informations and then generation of the Jenkins jobs. It seems that the author just uses Jenkins for the build. However, the system seems to be slow and has problems with synchronisation of the jobs. Also there are problems with the graphical side of the solution.

Some interesting ideas connected with the job generation are in the Shaw article [18]. The article also introduces the possibility of job generation from the templates and use of the Jenkins command line interface. Nevertheless, the article does not provide any examples of the templates or scheme how the system works.

Above we have mentioned several possibilities in the area of job generation. None of the approaches that were mentioned suits our needs. In our project we need to generate all kinds of jobs, as it is crucial to test the various aspects of the newly developed core. This includes the tests of various features that can be tied to very specific kinds of jobs. The approach mentioned in [17] seems to be interesting. For our use it appears to be too cumbersome. The lightweight solution with the command line interface would suit our needs better.

V. Job Generation

The main task that we need to address is the generation of the various jobs, which will ensure the complex testing of the core. As we plan to use the whole system also from the command line, we wanted to avoid the graphical interface, at least in the first version of the project. We may add the graphical interface in the later versions, but we need to keep the command line interface, as we would like to use the solution from the command line. This is also one of the reasons, why we can not use the plugins provided by Jenkins. They have very poor documentation and are primary focused for usage via the web interface.

The basic scheme of our system is illustrated in Figure 1. We can see that the whole system consists of just a few steps. The first part of the system is the sniffer. It works over the git repository in our case. Once the generation is triggered the job generator uses the templates to generate corresponding jobs. We will now give more detailed description of the aforementioned parts.

A. Sniffer

We called this part of the generation process a sniffer as it sniffs in the git repository for a new branches. The main

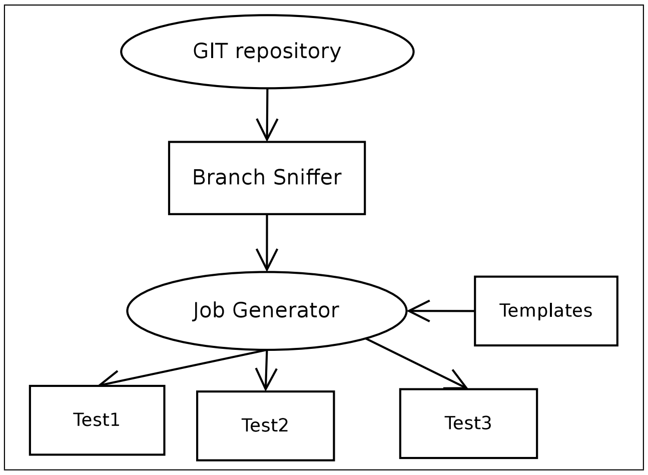

Figure 1. Scheme of the system.

role of the sniffer is to detect the creation of the new branch in given git repository and trigger the generation. The whole system is designed in a way that the sniffer can be replaced by a different component. In the future, we would like to add the support for other VCS. It also does not have to be present at all and can be completely removed. The generator can be started by a different tool, if it is compatible with the defined interface.

Though currently the role of the sniffer is to notify that the new branch has been created and deliver this information to the job generator. The sniffer has no further intelligence and the whole system is designed in a way that all the decisions should be made in the generator itself. In the latest version the sniffer has the shape of the unix script that is executed repeatedly by the operation system.

B. Templates

The second input into the job generator are the templates. We have various kinds of templates as we need to test various parts of the newly developed core. The main areas that has to be covered by test job generation are:

- compiler testing,
- functional verification,
- assembler testing,
- tools generation.

Please note that these are just the areas that needs to be covered, not the jobs. Under each domain there is a variety of jobs that are generated and later on executed. There is usually just one template per domain, just in case of the functional verification we need to have several templates, as this area is very vast and we were not able to stick to just one template.

As far as the templates itself are concerned, they are very simple and do not keep any intelligence. The intelligence, for example the name of the node, where the job will run is kept in the generator. The templates are in the XML format and are similar to the example in Section III. Consider for example that we want to generate the name of the node, where our job will be executed. The corresponding part in the template will have the following form:

```
<string>@NODE_NAME@</string>
```

C. Job generator

Now, when we described the inputs of the generator we will move to the generator itself. The job generator consists of several parts that are pictured in Figure 2.

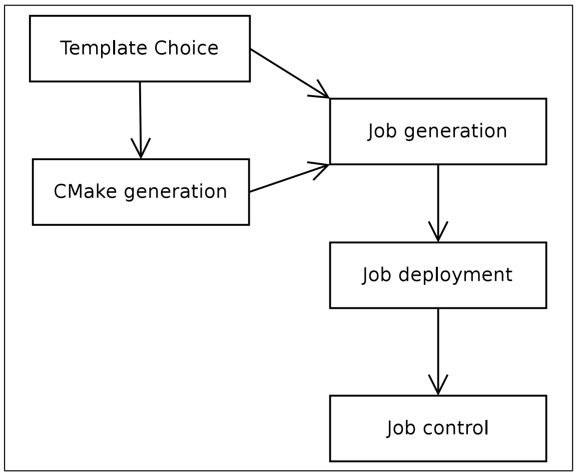

Figure 2. Scheme of the generator

We decided to implement the generator in Python language because it allows very fast development and the code is very easy to read and modifications are simple.

One of the first steps of the generation is the template selection. This part of the generator works over the configuration file that is present at the specific directory in the model branch that should be tested. We have proposed a simple format of the configuration file, which specifies the tested features. The other possibility we have is to automatically detect what features should be tested but we have chosen the configuration file, because some of the features can not be automatically detected. From the specification file we are able to determine what templates should be used. The specification file has two major tasks:

- define features that should be tested,
- specify parameters for the generators.

However, the automatic detection of the features that should be tested was not completely abandoned. The detection is present, but plays only the supplementary part.

Once the phase of the templates selection is finished we need to generate the CMake files that will fill into the templates the desired information. The generated CMake files are template specific as each template has different fields. Currently we generate one CMake file per template and we do so in the separate directories.

From the two above mentioned inputs, we can generate the job. The job generation is in fact just insertion of the data into the templates. We decided to do this via CMake, because it is one of the cleanest ways to do so. The most frequent facts that are generated are the following:

- branch used for testing,
- node, where the job is executed,

- bash script and the parameters,
- job name and view, where the job is placed.

The above mentioned information can be determined in the following way. The branch is one of the input parameters. It is delivered by the sniffer, but can be also delivered different way, it can be for example specified by the user.

The script that is executed could be the part of the template, however, this would increase the number of the templates significantly. Hence, we try to determine the name of the script. This could be done based on the information from the configuration. Some of the scripts may have variable number of parameters, but this we are able to determine from the directory structure of the model. Here we can see the supplementary part of the automatic detection.

The job name and view, where the job should be places, are also determined from the configuration file and repository name. We also plan in the future to use directory plugin in our installation, however, this should not be a problematic step.

The most complicated task is the selection of the correct node, where the job should be executed. There are certain jobs that can be executed only on the specific set of nodes. Typically this is true for the jobs that perform tests of the functional verification or tests of the synthesis. We have a special groups of nodes and special templates with the predefined sets of nodes. Nevertheless, for the majority of jobs we do not have to solve such issues. We keep a simple table of nodes, which is divided into the sections, which define what nodes are used for specific jobs. We choose the jobs with the smallest number of assigned jobs and optionally we modify the assigned value by hand.

There are also other information that can be filled into the template. But the four above mentioned are the most common ones. We have the predefined default values for all the parameters that would suit the most cases.

Very often we generate the parameters into the templates.

They are stored in the *parameters* section and later this parameters are used in the *builders* section. However, there are also parameters that are node dependent, or are defined globally in the Jenkins.

Very often the generated job needs to use the artifacts from the other jobs. Nevertheless, we try to keep the generator as lightweight as possible and do not want modify other jobs. The compatibility in this case is assured by the wildcards, and the name of the new job must fit into the wildcard. For example if the job is named Test-compiler-xxx the wildcard can be Test-compiler-*.

Once we have generated the jobs that are needed for the testing of the newly developed branch, we have to upload these jobs to the CI server. For this purpose we use the Jenkins command line interface that performs the job upload and also registers the job.

VI. RESULTS

With the current implementation of the simple job generator we have performed several tests. We have tried to generate the set of tests that are typical for our project. The tests are divided into two sets. The basic set consists of tests that test compiler and assembler and full set adds also tests for functional verification. The templates that are needed for generation of such tests were added into the template set. The basic set consists of three jobs and full set consists of 12 jobs. We have set the polling time to 6 minutes, so every 6 minutes is the VCS server polled for the new branches.

The times needed for the generation are summarised in the following table. We have performed 10 different runs: five for basic set of tests and five for the full set of tests. The last run was triggered manually.

TABLE I. COMPARISON OF GENERATION TIMES.

Run	*Basic set*	*Full set*
1	124s	516s
2	248s	524s
3	194s	212s
4	150s	317s
5	91s	412s
Manual run	42s	178s

We can see in the Table I that the generation of the three jobs takes 42 seconds, which gives exactly 14 seconds per job. When we try to generate the full set of 12 jobs, it takes 178 seconds. That is approximately 15 seconds per job. All of the jobs we generate are multiconfiguration jobs. The generation times vary for the basic set from 91 to 248 seconds. That is perfectly accurate, as the delay caused by the front end is up to 360 seconds. The generation of the full set is also affected by the front end and should be from 178 seconds up to 538 seconds. Our measurements confirm that.

We have also tried to create the jobs manually. The group that created the jobs consisted of two persons. We tried to create the basic set of testing jobs, and then the full set of jobs. The basic set of tests include the generation of three jobs and covers the compiler and assembler. The full set of jobs contains also jobs for verification. Together this set contains 12 jobs. Hence, the sets are the same as in the previous measurement.

TABLE II. COMPARISON OF CREATION TIMES.

Method	*Basic set*	*Full set*
Lissom Generator	182s	499s
Manual creation	486s	2197s

In the Table II we can see that the manual creation of the jobs was very slow in comparison with the generator. Especially in case we have to create the set of 12 jobs the task was very time consuming.

The last comparison we made was with the Jenkins job generator plugin. We used the Jenkins server in version 1.656 and the plugin was in the version 1.22. The Jenkins server was running on the server with the 4 cores Intel i5 and has 8 GB of the memory. The same set of jobs as above was generated.

TABLE III. COMPARISON OF CREATION TIMES.

Method	*Basic set*	*Full set*
Lissom Generator	103s	361s
Jenkins generator plugin	148s	839s

The results are gathered in the Table III. It is clear, that Lissom generator was fastest in both tested cases. However, in case of generation of just three jobs, the times were comparable. And in case of maximal 360 seconds delay, the Jenkins job generator can be even faster. Nevertheless, in case of generation of the big set the generator had clear advantage even in case of maximal delay caused by the front end. Moreover when compared to the times without delay, the speed of Lissom job generator can not be matched.

Other advantage of the job generator is the fact that it is very lightweight and can be used for any kind of jobs. This largely depends on the templates that will be created.

VII. Conclusion

In this paper, we sketched the simple generator of the Jenkins jobs that would suite our needs in the Lissom project. We need the generator that can be started by various ways is lightweight and can generate all kinds of jobs. This was one of the basic requirements that was not met by any plugin that is currently available for Jenkins. We also wanted the tool to be at least partly independent of Jenkins as it is not rare that the plugins do not cooperate well.

The current implementation of our generator is dependent just on the internal representation of the job. This is not a problem, as it is very simple to deploy new templates. At the same time, the internal job representation is not likely to change as it would imply the changes in all plugins currently used by Jenkins.

We also put the generator under the tests and the gathered results are very positive. As far as the speed of the generator is concerned it can not be matched by any tool that is currently available. In the future we would like to add to the generator also other functionality such as work with the directory plugin and also ability to register the jobs for artifact download.

We created a tool that helps us to generate new sets of test every time the new core is developed. It gives us the higher level of test automation.

Acknowledgment

This work was supported by The Ministry of Education, Youth and Sports of the Czech Republic from the National Programme of Sustainability (NPU II); project IT4Innovations excellence in science - LQ1602.

References

[1] G. De Micheli and W. Rolf, E.and Wolf, Readings in Hardware/Software Co-design. Morgan Kaufmann, 2001, ISBN: 9781558607026.

[2] "Texas Instruments," http://www.ti.com/general/docs/newproducts.tsp (July 2016), 2016.

[3] F. Oquendo, "π-adl: an architecture description language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures," ACM SIGSOFT Software Engineering Notes, vol. 29, no. 3, 2004, pp. 1–14.

[4] J. Teich, "Hardware/software codesign: The past, the present, and predicting the future," Proceedings of the IEEE, 2012.

[5] Lissom, "Project Lissom Webpages," http://www.fit.vutbr.cz/research/groups/lissom/ (August 2014), 2014.

[6] K. Masarik, "System for hardware-software codesign," Master's thesis, Faculty of Information Technology, Brno university of Technology, 2008.

[7] K. Suzuki, S. Mita, T. Fujita, F. Yamane, F. Sano, A. Chiba, Y. Watanabe, K. Matsuda, T. Maeda, and T. Kuroda, "A 300 mips/w risc core processor with variable supply-voltage scheme in variable threshold-voltage cmos," in Custom Integrated Circuits Conference, 1997., Proceedings of the IEEE 1997. IEEE, 1997, pp. 587–590.

[8] B. Smith, "Arm and intel battle over the mobile chip's future," Computer, vol. 41, no. 5, 2008, pp. 15–18.

[9] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, "The risc-v instruction set manual, volume i: Base user-level isa," EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62, 2011.

[10] J. Kroustek, "Retargetable analysis of machine code," Master's thesis, Faculty of Information Technology, Brno university of Technology, 2014.

[11] L.-T. Wang, Y.-W. Chang, and K.-T. T. Cheng, Electronic design automation: synthesis, verification, and test. Morgan Kaufmann, 2009.

[12] M. Fowler and M. Foemmel, "Continuous integration," Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf, 2006, p. 122.

[13] G. Booch, Object-oriented Analysis and Design with Applications (2Nd Ed.). Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1994.

[14] Jenkins, "Jenkins website," https://jenkins.io/ (July 2016), 2016.

[15] "Template Project Plugin," https://wiki.jenkins-ci.org/display/JENKINS/Template+Project+Plugin (July 2016), 2016.

[16] "Job Generator Plugin," https://wiki.jenkins-ci.org/display/JENKINS/Job+Generator+Plugin (July 2016), 2016.

[17] F. Lier, J. Wienke, and S. Wrede, "Jenkins for flobi–a use case: Jenkins & robotics," in Jenkins User Conference, 2013.

[18] K. Shaw, "Generating New Jenkins Jobs From Templates and Parameterised Builds," http://www.blackpepper.co.uk/generating-new-jenkins-jobs-from-templates-and-parameterised-builds/ (July 2016), 2012.