Combining Model-based Testing and Continuous Integration

Martin Koskinen, Dragos Truscan, Tanwir Ahmad and Niklas Grönblom
Åbo Akademi University,
Turku, Finland

Email: [martin.koskinen, dragos.truscan, tanwir.ahmad, niklas.gronblom]@abo.fi

***Abstract*—We present our approach on combining model-based testing with the continuous integration process. The main benefits of this combination lie in the ability to automatically check the conformance of the implementation with respect to its specification, while shortening the feedback cycles and providing increased test coverage. A case study on developing an in-house academic tool is presented in which the online model-based testing approach is used with the continuous integration process.**

***Keywords*—Continuous Integration; Model-Based Testing; UPPAAL Timed Automata**

I. Introduction

Continuous Integration (CI) is a software development practice in which developers frequently integrate their work [1] [2]. The CI process runs continuously during the lifetime of a project, resulting in that the different parts of the product are always updated, integrated, and tested. Regression test suites are used for checking the quality of the integration.

One perceived problem of CI is that the increasing size of the code base leads to increasing run time of the integration build. According to Rogers [3], one of the main causes behind this, is not the increasing compilation time, but rather the increasing number of tests executed. In addition, the maintenance of the regression test suites can be time consuming and error-prone. As current practice, Duvall et al. [2] recommend that system builds are run several times, or at least once a day, which imposes tight constraints on the CI and testing process.

In this paper, we discuss the inclusion of the model-based specifications and automated test design techniques into the CI process, in order to enable incremental development, shorter feedback cycles and increased test coverage. There are two enablers for achieving these targets: (a) early detection of errors is facilitated by performing simulation and verification on the model-based specification after each update of the specifications; and (b) the test suite corresponding to the latest version of the specifications is generated automatically and made available to the CI process.

Our testing approach focuses on the conformance testing using automated test generation techniques. The system is developed incrementally. The specifications are done using *UPPAAL timed automata* (UPTA) [4]. Every time a new feature is added, it is first modeled, simulated and verified. Once the specifications are updated, they are used for automated test generation. When the feature is also implemented in the source code, the automatically generated tests are executed in order to detect possible behavioral inconsistencies between the specification and the implementation, and a report is issued as feedback.

The paper has the following structure: Section II will briefly discuss different background concepts. In Section III we introduce a generic process for combining MBT with CI, followed by a concrete case study in Section IV. Section IV also describes how we applied this approach in a practical software development project. An evaluation of our approach is discussed in Section V, whereas final thoughts and future work are presented in Section VI.

II. Background

In the following subsection, the CI process is described in more detail, followed by a short introduction to *Model-based testing* (MBT) [5]. The last subsection briefly introduces the UPPAAL tool and its capabilities.

A. Continuous Integration revisited

The traditional workflow of the CI process can be summarized as following. When a developer has finished an implementation task, he makes a local build to see whether the program builds correctly. Ideally, he also runs tests locally to verify that the implementation is correct. After this, the developer commits the code to the *Source Code Management* (SCM) system.

A CI-server is used to integrate source code from different SCMs used in the process and to create an integration *build*, either at regular time intervals or based on commit triggers linked to the SCMs. The build process might contain different kinds of code analysis, for example to ensure that the code conforms to common code conventions, or integration/acceptance tests for the newly built software. Subsequently, feedback is provided to the concerned parties on the outcome of the build. If errors occurred or conventions were violated, these are mentioned in the report. When errors or violations are detected in the integration build, the responsible developer is supposed to fix them as soon as possible. In many instances of CI, the CI process is stopped, i.e., no one can commit updates, before the previous failed build is fixed or the code is reverted to the previous working version.

 ISBN: 978-1-61208-304-9

B. Model-based Testing

MBT is a testing approach which reduces the effort needed for testing [6], by automatically designing test suites from abstract behavioral specifications of the *system under test* (SUT). The main philosophy behind MBT is to automatically generate tests from abstract models, which specify the expected behavior of the system under test. Based on how tests are generated and executed MBT has two flavors: online and offline [7]. In *online* testing, tests are generated from the model and executed on-the-fly against the SUT. At each step, a new test is designed based on the output of the previous test. In contrast, in *offline* mode, all test are pre-generated (scripted) into an executable format, which is then executed in batch mode using test execution frameworks.

C. UPPAAL

UPPAAL is a toolbox for verification of real-time systems [4]. The tool provides a graphical user interface for editing, simulating and verifying models based on an extended version of time automata, referred to as UPTA [4].

Informally, in UPTA, systems are modelled as a network of timed automata which communicate with each other through global variables and channel synchronizations. An edge, that connects locations, can be decorated by a guard, allowing or not allowing the edge to be taken, depending on some condition. A channel can be sending or receiving synchronizations, which are annotated by the suffixes *!* and *?* subsequently. An edge with a sending synchronization requires one edge that can receive the synchronization. If several receiving channels are available, one will be chosen non-deterministically. Synchronization channels can be declared as broadcast channels, which removes the requirement of a synchronization receiver. This implies that broadcast synchronizations will be sent, even if there is no receiver. If there are several receivers available for a broadcast, all receivers will receive the synchronization simultaneously. On edges, variable values are updated by assignment statements. Initial locations are marked with a double circle. There are two other special location types except the normal location. An urgent location, which stops time, is marked by an 'U'. A committed location, marked by a 'C', is stricter than the urgent one, since the automaton is allowed to leave the location in the next transition without intervention by another process. For a formal definition of UPTA, one could refer to [4].

III. Combining MBT and CI

As mentioned in the introduction, in order to take advantage of the MBT approach, we integrate MBT into the CI process. We use MBT to make sure that the specification and the implementation of the SUT always conform to each other. A generic view of our CI process is shown in Figure 1.

The CI process employs several SCM servers, used for maintaining different artifacts of the development process. In Figure 1, there are different SCMs for versioning the source code, specifications, toolchain, test suites, etc. In practice, one can use the same SCM server for accommodating several or all artifacts.

Several teams are involved in the development, for instance, a specification and a development team, each following specific processes and committing regularly (with different frequency) to the corresponding SCM server. Basically, when a new feature is introduced, it is specified, validated and then committed. Validation helps in detecting potential inconsistencies in the specifications, such as misunderstanding of requirements or omissions. The simulation and validation ensure that the desired behavior can be achieved.

The task of the development team is to implement the requested features according to the specifications. The developers can test their code locally, after which they commit it to the corresponding SCM server. These tests may be unit tests developed by the developers themselves or tests retrieved from the test suite SCM.

As the main idea of software testing is to verify the behavioral conformance between the specification and the implementation, every time one of them is updated, we check that they conform to each other using MBT.

This process is controlled by a CI server, which is configured to monitor the SCM repositories involved in the build. Whenever a commit to any of the repositories is detected, all repositories related to the project are updated on the CI server. Regardless of which repository triggered the update, the following steps are executed depending whether an offline or online testing approach is used.

a) Offline process: When a build is triggered due to changes on a SCM server storing the specifications, the CI server checks if the existing test suite needs updating due to changes to the specifications. If needed, a fresh test suite is generated from scratch. The test generation replaces the need for maintaining and deciding which tests should be added or removed from a manually created regression test suite. The updated test suite is stored on a SCM server for later use. The test suite is generated once and can be reused as long as the specifications do not change. Updating the test suite can be done as part of the CI, or as a separate process as for convenience is showed in Figure 1.

When the code is updated on the corresponding SCM server, a build is started. Upon completion, the test suite is executed in batch mode against the SUT. Finally, the developers get feedback on how the build proceeded.

b) Online process: For distinction, the online testing process is depicted with thicker line pattern in Figure 1. When a build is triggered by either of the SCM servers, the CI server starts by building the project. As in online mode the tests are generated and executed on-the-fly, there is no previous test suite as such to be updated.

IV. Case Study

In this section, we provide a concrete example of how the generic process described in Section III can be put into practice. The case study presented in this paper is part of the development process of an academic tool for performance

 ISBN: 978-1-61208-304-9

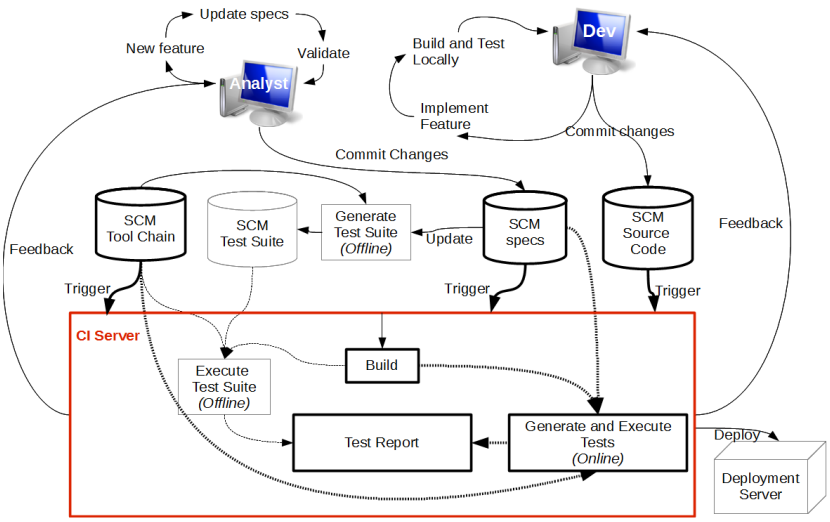

Figure 1. Generic process overview

testing, called MBPeT. MBPeT [8] is a model-based load generation tool which uses probabilistic models to generate load and applies it interactively against the target system. The development team consists of 2-4 developers, while the specification team consists of 1-3 persons.

In the following subsections, we briefly describe different activities involved in our development process.

A. Model-based specifications

The MBPeT tool has a distributed architecture, in which one master node controls several slave nodes that are actually generating the load by executing the desired number of concurrent virtual users. During load generation, the master node decides how the load is distributed to the slaves. Each slave has a predefined saturation threshold for the local resources which is used to ensure that the slave node is able to generate the required load. Whenever the saturation threshold is reached, the load on the current slave is kept constant, while the remaining load is delegated to the next available slave.

In the specification phase, the models of both the master node (see Figure 2) and the slave node (see Figure 3) are created and their communication is modelled in UPTA. The behavior of the two node types is described in the following:

c) Master model: The master process in Figure 2 is designed to handle several slaves. It starts by waiting for all slave processes to connect, by receiving the *i_slave_connect* synchronization from each slave process. Then the master process continues with configuration and initialization of the connected slaves, by sending an *o_initSlave* synchronization to each slave process. The initialization is completed when all slave processes have sent an *i_slaveInit* synchronization to the master. At this point, the master requests the first slave to start load-generation by sending an *o_generateLoad* synchronization to it. The master continues to listen for either an *i_slaveSaturated* or an *i_slaveDone* synchronization. If an *i_slaveSaturated* is received and there are no available slaves to start, the `failure` variable is set to *1*. When the master process has received the *i_slaveDone* synchronization from all started slaves, the master proceeds by shutting down all connected slaves, by sending an *o_killSlave* synchronization to each slave process. At the end of the test session, the master process enters the *STOPPED* location.

d) Slave model: The process model, corresponding to the previously described master process, is shown in Figure 3. The process starts by sending an *i_slave_connected* signal to the master process. The master initializes the slaves by sending configuration information, which corresponds to the slave process reaching the *Initialized* location. Here it waits for either an *o_killSlave* or an *o_generateLoad* synchronization. The former results in a return to the initial location and the latter instructs the slave process to start generating load. The slave saturation is calculated by looping via the locations *Load_calculated* - *Saturation_Check*. If the slave's load variable is greater than a *threshold* value, the slave is considered saturated. When the slave becomes saturated, it transitions to the *generate_load_saturated* location by sending an *i_slaveSaturated* synchronization. When test time runs out, the slave transitions to the initial location via the *Load_generation_completed* location. If the test duration runs out without the slave being saturated, the slave transitions to the *Load_generation_completed* location via location *TestDuration_timeout* and sends an *i_slaveDone* synchronization to the master. The slave instances share a global clock *timer*, which is used for exiting the load generation when test time runs out. The clock is reset when the first slave starts generating load. For the rest of the slaves, the load generation is started without resetting the timer. The models discussed above allow for an instance of the master to communicate with several instances of the slaves, thus imitating the architecture of the real tool.

Simulation The models of the MBPeT tool have been created incrementally, one feature being added at the time.

 ISBN: 978-1-61208-304-9

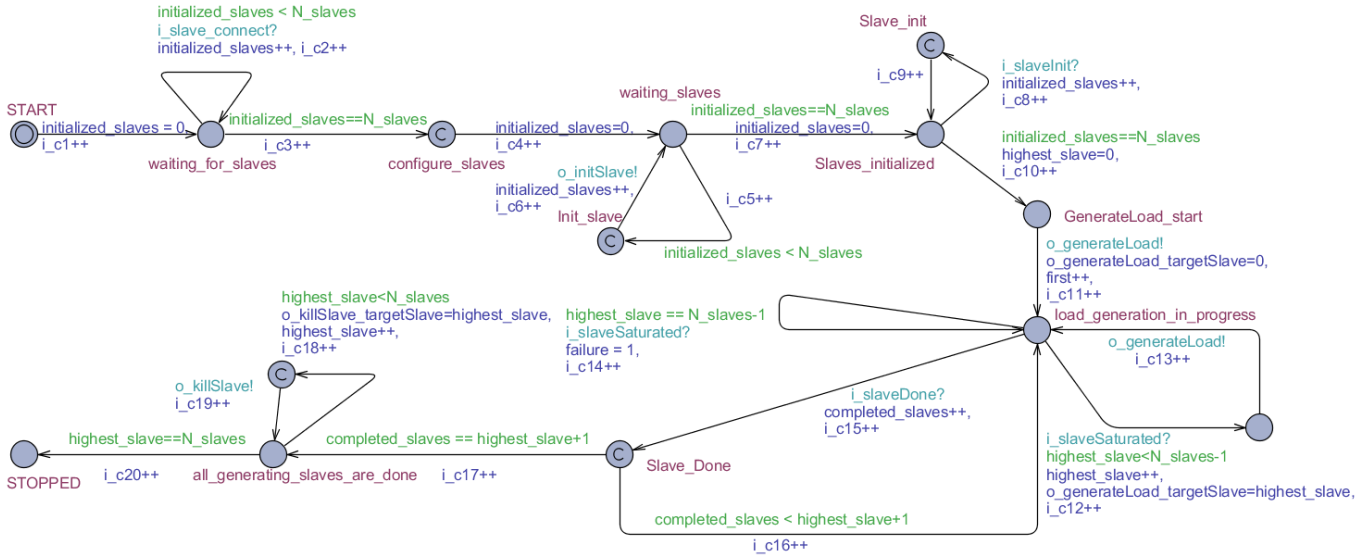

Figure 2. Master process model

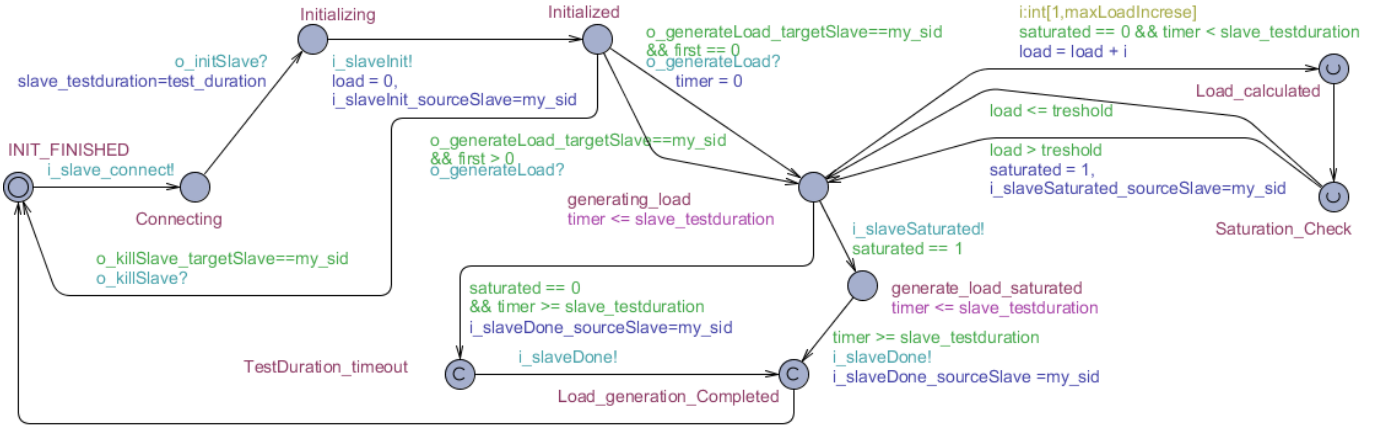

Figure 3. Slave process model

After each feature is modeled, the models are simulated in the UPPAAL tool, allowing us to experiment and check if the proposed models behave as specified in the requirements.

Verification The models are also used for verification of different system properties, e.g., reachability, safety, and liveness. UPPAAL provides its own verification engine [4], which uses a simplified version of the TCTL language [9]. For brevity, we refer the reader to [4] for more details.

B. CI process

In our development process, we have used three repositories: one for the implementation of the SUT, one for the specifications of the SUT and one for the tool chain. We follow the online testing process described in Section III.

1). When a new feature is to be introduced to the MBPeT tool, it is first modelled in UPTA. The specification is simulated and verified. When the new feature is considered ”approved”, the new models are submitted to the specifications SCM. The first time, this commit will trigger a build which will fail, since the specification contain an unimplemented feature. The failed build shows that the system implementation is lagging behind the specifications, i.e., it does not conform to the specified behavior.

2). The development team will start implementing the new feature in code. When the implementation is ready, the developers run unit tests locally. The development team also has access to the entire testing tool chain to validate their updated implementation before committing. When the code is committed, a new build is started followed by the model-based testing of the build. If the integrated system behaves according to the specification, the build will pass the testing successfully; otherwise an error-report will be generated.

3). Both the development and the specification teams will receive an error-report, which will be discussed and analyzed in order to detect the source of the failure(s). It may happen that the error is in the specification, instead of the implementation, due to misunderstanding of the requirements or undetected

 ISBN: 978-1-61208-304-9

errors in the specification. The error may also be located in the tool chain or in the test environment.

4). The identified failures are fixed by the team responsible. Upon committing the updated artifact, a new build is triggered which should result in a successful test run.

The CI process is supported by the Jenkins CI server [10], an open source continuous integration server, configurable via a Web interface. Its functionality is extendable via plugins, e.g., integration with SCM systems. The building of projects is configured via *jobs*. A job can be triggered manually, based on a time trigger or based on an event, e.g., the completion of another job.

The SCM software we use is Subversion [11]. In order to implement the job-triggering mechanism, we implemented a set of ”hook” scripts, which are run on the SCM servers and monitor certain paths for commits. When a commit is detected by a hook script, via a regular expression match on the monitored path, the Jenkins CI server job is triggered by an a HTTP request.

C. Test generation

In our approach, we have targeted the online mode of MBT, since it addressed better the non-determinism in the specifications. In our case, study we have non-determinism at several locations, for example in the *generating_load* location. There is no limit on how many times, if any, the loop calculating the slave's saturation is taken.

Since the models of both the master and the slave nodes are created and verified in the specification phase, any one of them can be used as a SUT model, whereas the other one will be used as environment model. Consequently, in our CI environment, we have two independent jobs: one considering the master node as the SUT, while the other uses the slave node as the SUT. In this paper, we selected for exemplification the setup where the master node is the SUT, and the slave nodes act as the test environment.

An overview of our model-based test setup, is shown in Figure 4. We use three repositories: specifications, source code, and tool chain. Whenever one of the repositories is updated the commit trigger is activated and the CI process is started by updating the repositories (if new versions are available), building and deploying the implementation, instantiating the tool chain and starting test execution.

The tool chain is composed of several components. *Distributed TRON* (DTRON) [12] is a tool for distributed online test generation. DTRON uses a Spread network, the Spread Toolkit [13] for its multicast communication between different instances of DTRON and an eventually distributed SUT. In order to interface with the *implementation under test* (IUT), an *adapter* written in Java, is used to convert tests messages received from DTRON into messages compatible with the communication protocols required by the IUT. DTRON will receive output from the SUT via the adapter, which distributes it via the Spread network. The received values are compared with the expected output and a verdict is given.

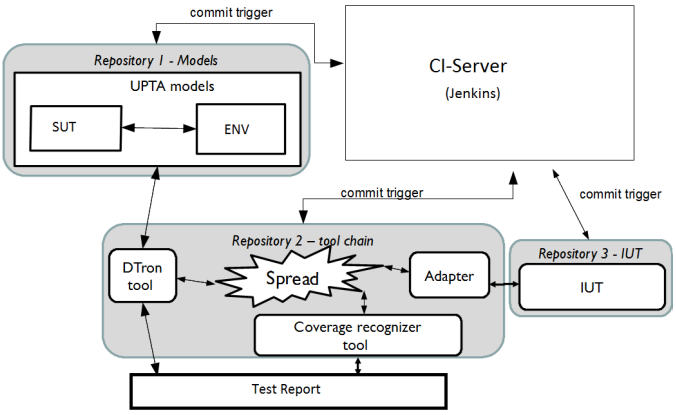

Figure 4. Overview of the test setup

The adapter is updated every time new observable interfaces of the SUT are added to the UPTA specifications. There is a naming convention for making channels and variables observable by DTRON. A channel name prefixed by *o_* means the channel is used for IUT-to-model communication. Similarly, a channel prefixed by *i_* is used for model-to-IUT communication. Integer variables can be sent along with these synchronizations. In this case, the variable name is prefixed by the observable synchronizing channel's name, i.e., *i_channelname_variablename*; see Figures 2 and 3 for exemplification.

Once the test session is started, DTRON will generate tests via symbolic execution of the specifications using randomized choice of input. The observable communication between the environment model and the system model is captured by the adapter and send or expected to be received from the SUT. Whenever an expected output is not received from the IUT with the expected value or within the specified timeframe, a failure will be observed and the test generation and execution will be stopped. Consequently, the build job on the CI-server will generate a test report and will send it to the respective teams.

D. Measuring coverage

In our approach, for each new commit, we measure both specification coverage and code coverage for each test run.

Specification coverage. In order to achieve a certain coverage level, with respect to specific coverage criteria, we have two options available when using UPPAAL-based tools. The first option is to use an environment model which will drive the test generation to follow specific test targets in the SUT model, as described by Hessel et al. [14]. The second option is to have an environment model which does not enforce explicitly specific test targets in the SUT model (e.g., the model of the slave node) and to recognize the coverage level of the test run upon completion.

In our CI process, we use the second approach, which requires an additional utility script to be included in the process. The general idea is to automatically customize the UTPA models, without modifying the original behavior, in

 ISBN: 978-1-61208-304-9

order to allow one to observe how different structural parts of the model have been covered during the test execution.

The *coverage recognizer tool* (CRT), as shown Figure 4, has two main functionalities. When the specification is updated and detected by the CI-server, the script processes the UPTA model by adding unique counter variables and a corresponding updates statements on each edge of the SUT model. See for exemplifications variables $i\_c1, \ldots, i\_c20$ added to the Master model in Figure 2.

An observable channel is also added, hereafter referred to as the *counter channel*, which is used for synchronizing the counter variable values to the CRT. The channel is declared of type *broadcast*, which is weakly synchronized and therefore does not require a receiver automaton [4]. In order to be visible on the Spread network, the counter variables and the counter channel follow the naming conventions of DTRON as explained earlier. The counter channel has to be synchronized at some point for the CRT to be able to produce a coverage report. To achieve this, the tool adds a process to the system, containing one location with a self-edge, that synchronizes the counter channel periodically.

The second functionality of the CRT is to connect to the Spread network and monitor the counter variables and to build statistics about the edges visited during the test run. At the moment, CRT provides support for *edge*, *edge-pair* and *requirement coverage*, respectively. However, other structural coverage criteria could be implemented in the tool. The requirement coverage criterion, is a simplified version of the edge coverage one described above, in which test targets fulfilling certain system requirements are manually added to the model as counter variables.

If we would follow an offline test generation approach, a set of traces satisfying, e.g. edge coverage, can be easily obtained via model-checking of the property $E <> i\_c1 \&\& \ldots \&\& i\_c20$ using tools like *verifyta* provided by UPPAAL as described in [14].

Code coverage. Since we have access to the source code of the IUT we also track how much of the code has been covered by each test run. For this purpose, we use the *coverage* tool for Python [15]. The tool counts the number of statements in the source files and monitors which of them are executed. At the end of the test run, it provides a coverage report detailing the coverage level for different source files.

V. Evaluation

MBT has two distinct components: modeling and test generation. Each of them brings its own benefits to reducing the effort of the testing process. The main benefit of modeling is that it forces the designers to simulate and verify the system specification before deciding to implement a new feature. During the specification and development of the MBPeT tool, we detected many specification inconsistencies which could have resulted otherwise time spent during the implementation, testing and debugging.

The automatic test design has also its benefits, even if the test suite has to be generated for each iteration. Due to the online generation approach and also of the available tool support, generating only parts of the test suite when the model changes has not been considered. We do not consider it a problem for two reasons: 1). with our approach the duration of a test run requires less than one minute to achieve an acceptable edge coverage level and 2) if the models would become too complex to handle timely test generation, then raising the level of abstraction or focusing the test generation on certain parts of the models will help.

However, if complex test suites cannot be avoided there exist a body of work which has addressed regression testing in the context of MBT, e.g., [16], [17], [18], in its vast majority targeted to offline test generation. In addition, there exist already commercial tools such as Conformiq tool-set [19], which optimizes the offline test suite generation by generating only new test cases and removing old test cases which are not relevant anymore.

With respect to our case study, the code base of the Master node is approx. 2100 LOC written in Python, whereas the test adapter needed for the models in Figures 2 and 3 is slightly over 200 LOC written in Java.

Letting the DTRON tool randomly generate tests from the model in Figure 2, we could identify six different test scenarios, as depicted in Table I. For each scenario, we extracted the corresponding edge coverage and statement coverage levels. As shown in this table, the minimal edge coverage achieved for our particular models is 70%, when running with one slave which does not saturate. This corresponds to 91% statement coverage. The highest edge coverage, 95%, is achieved when having more than two slaves, of which at least one saturates, one generates load unsaturated, and one is idling. In this case, the statement coverage increased to 98% coverage.

With the model in Figure 2, full edge coverage cannot be achieved due to two mutually exclusive paths in the model: for a given set of slaves one cannot have both an idling slave (which would be killed via edges `c18`, `c19`) and all slaves saturated (edge `c14`).

Test Scenario	Covered Edges	EC	Statement Coverage
1 slave, no saturation	1-11, 15, 17, 20	70%	91%
1 slave, saturation	1-11, 14, 15, 17, 20	75%	91%
2 slaves, slave 1 saturated	1-13, 15, 16, 17, 20	85%	92%
2 slaves, both saturated	1-17, 20	90%	91%
2 slaves, 1 idle	1-13, 15, 17-20	90%	92%
>2 slaves, 1 idle	1-13, 15-20	95%	98%

TABLE I
Coverage results

Due to the way the tests are generated from these models, with each test run we may obtain a different trace depending on the number and behavior of the slaves. However, using two slave nodes in the test configuration, provided an acceptable edge coverage level. Adding another slave will increase the edge coverage by 5% and the statement coverage by 6%. At the moment, we did not consider this approach necessary, since when inspecting the source code coverage for all three test

 ISBN: 978-1-61208-304-9

scenarios using two slave nodes, we found that actually the entire code base was covered by the respective traces.

VI. Conclusion and Future Work

We have presented an approach in which model-based testing and continuous integration approaches have been combined in order to lessen the testing effort and consequently shorten the integration cycles.

Having performed simulation and verification of the specifications increased their quality and decreased the number of failures originating in the specifications, such as common mistakes, omissions and misinterpretations of the requirements. The UPTA formalism allowed us the modeling of time and the verification of time properties. Using automatically generated tests decreased the time spent to develop tests every time a new update was performed either in the specification or in the implementation.

Since the repositories can be updated independently, the modeling and development teams are immediately aware of problems in the build. Ideally, the implementation and the models should be in sync, that is the implementation should reflect the model. As long as the tests conclude that the implementation and models conform to each other the builds are successful. If they start diverging, we can conclude that either the model or implementation is erroneous, or the other team has not yet updated their part of the system to conform to new requirements.

In our current case study, we used tests generated and executed on-the-fly. This approach has both advantages and disadvantages. As explained in the paper, one benefit is that using online MBT allows for non-deterministic behavior due for instance to concurrency or to time/value domains. In addition, it does not require an additional test execution framework to be included in the toolchain, although it does require the implementation of the adapter. However, the adapter has to be updated only when new observable interfaces are added to the SUT, otherwise it can be reused as such.

Among the perceived drawbacks of online MBT is that tests have to be regenerated from scratch every time, which can be time consuming. However, since all the tests are generated automatically, the generation times are short for reasonably sized models. If the models become too complex, increasing the level of abstraction or focusing only on certain parts of the functionality should be considered. For instance, with our models, the average test run is on average less than a minute. This means that one can get a test report in several minutes since a new version is committed to the SCM servers. Another drawback of our online MBT approach is that the test session is stopped on the first failure of the tests, which compared to offline testing will not give a good overview of the failed/passed test case ratio. However, observing the achieved coverage with the CRT tool alleviated this problem and allowed us to identify which parts of the specification passed testing and which did not.

Future work will look into more detail at using offline MBT, and, in particular in deploying more efficient methods of model and test suite update via the modularization of test specifications. Also, by improving test reporting, more meaningful debug information can be provided for the development teams.

References

[1] M. Fowler. (2006, May) Continuous integration. Retrieved: 20.08.2013. [Online]. Available: http://martinfowler.com/articles/continuousIntegration.html

[2] P. M. Duvall, S. Matyas, and A. Glover, *Continuous integration: improving software quality and reducing risk*. Addison-Wesley Professional, 2007.

[3] R. Rogers, "Scaling continuous integration," in *Extreme Programming and Agile Processes in Software Engineering*, ser. Lecture Notes in Computer Science, J. Eckstein and H. Baumeister, Eds. Springer Berlin Heidelberg, 2004, vol. 3092, pp. 68–76. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-24853-8_8

[4] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on UPPAAL," in *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, ser. LNCS, M. Bernardo and F. Corradini, Eds., no. 3185. Springer–Verlag, September 2004, pp. 200–236.

[5] M. Utting and B. Legeard, *Practical Model-Based Testing – A Tools A pproach*. San Francisco, CA, USA: Morgan Kaufmann, 2007.

[6] ITEA 2, "D-MINT project result leaflet: Model-based testing cuts development costs," http://www.itea2.org/project/result/download/result5519?file=06014_D_MINT_Project_Leaflet_results_oct_10.pdf, February 2010, retrieved: 20.08.2013.

[7] G. J. Myers *et al.*, *The Art of Software Testing*. John Wiley & Sons, Hoboken, NJ, 2nd ed edition, 2004.

[8] T. Ahmad, F. Abbors, D. Truscan, and I. Porres, "Model-based performance testing using the MBPeT Tool," Turku Centre for Computer Science, TUCS Technical Reports 1066, 2013.

[9] R. Alur, C. Courcoubetis, and D. Dill, "Model-checking for real-time systems," in *Logic in Computer Science, 1990. LICS'90, Proceedings., Fifth Annual IEEE Symposium on Logic in Computer Science*. IEEE, 1990, pp. 414–425.

[10] Jenkins CI - Meet Jenkins. Online at https://wiki.jenkins-ci.org/display/JENKINS/Meet+Jenkins. Retrieved: 20.08.2013.

[11] Subversion. Online at http://subversion.apache.org/. Retrieved: 20.08.2013.

[12] A. Anier and J. Vain, "Model based continual planning and control framework for assistive robots." in *PECCS 2012 - Proceedings of the 2nd International Conference on Pervasive Embedded Computing and Communication Systems*, C. Benavente-Peces, F. H. Ali, and J. Filipe, Eds. SciTePress, 2012, pp. 403–406. [Online]. Available: http://dblp.uni-trier.de/db/conf/peccs/peccs2012.html

[13] The Spread Toolkit - Overview. Online at http://spread.org/SpreadOverview.html. Retrieved: 20.08.2013.

[14] A. Hessel, K. G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou, "Testing real-time systems using UPPAAL," in *Formal methods and testing*, R. M. Hierons, J. P. Bowen, and M. Harman, Eds. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 77–117. [Online]. Available: http://dl.acm.org/citation.cfm?id=1806209.1806212

[15] Code coverage measurement for Python – coverage, v. 3.6. Online at https://pypi.python.org/pypi/coverage. Retrieved: 20.08.2013.

[16] B. Jiang, T. Tse, W. Grieskamp, N. Kicillof, Y. Cao, and X. Li, "Regression testing process improvement for specification evolution of real-world protocol software," in *Quality Software (QSIC), 2010 10th International Conference on*. IEEE, 2010, pp. 62–71.

[17] E. Fourneret, F. Bouquet, F. Dadeau, and S. Debricon, "Selective test generation method for evolving critical systems," in *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*. IEEE, 2011, pp. 125–134.

[18] Y. Chen, R. L. Probert, and H. Ural, "Model-based regression test suite generation using dependence analysis," in *Proceedings of the 3rd international workshop on Advances in model-based testing*, ser. A-MOST '07. New York, NY, USA: ACM, 2007, pp. 54–62. [Online]. Available: http://doi.acm.org/10.1145/1291535.1291541

[19] Conformiq tool set. Online at http://www.conformiq.com/. Retrieved: 20.08.2013.

 ISBN: 978-1-61208-304-9