Model-based Testing of Stateful APIs with Modbat

Cyrille Artho*, Martina Seidl†, Quentin Gros‡, Eun-Hye Choi*,
Takashi Kitamura*, Akira Mori*, Rudolf Ramler§, and Yoriyuki Yamagata§
*AIST/ITRI, Amagasaki, Japan
{c.artho,e.choi,t.kitamura,a-mori,yoriyuki.yamagata}@aist.go.jp
†Johannes Kepler University, Linz, Austria
martina.seidl@jku.at
‡University of Nantes, Nantes, France
quentin.gros@etu.univ-nantes.fr
§Software Competence Center Hagenberg, Hagenberg, Austria
rudolf.ramler@scch.at

***Abstract*—Modbat makes testing easier by providing a user-friendly modeling language to describe the behavior of systems; from such a model, test cases are generated and executed. Modbat's domain-specific language is based on Scala; its features include probabilistic and non-deterministic transitions, component models with inheritance, and exceptions. We demonstrate the versatility of Modbat by finding a confirmed defect in the currently latest version of Java, and by testing SAT solvers.**

Keywords*—*model-based testing; software test tools; domain-specific language; extended finite state machines; component-based systems; exception testing

I. Introduction

Model-based testing derives test cases from an abstract model of the system under test and/or its environment [1]. The tool Modbat presented in this paper supports model-based testing by providing a domain-specific language (DSL) to define high-level models in a user-friendly way. Extended finite-state machines (EFSMs) combine a high-level model with extra variables and transition functions [1]. Test cases are generated as sequences of method calls to the application programming interface (API) of the system under test (SUT). Results can be checked using assertions, or stored in model variables, to be used in subsequent calls.

Our DSL significantly reduces the notational overhead compared to other tools [2]. Exceptions are managed on the model level, which increases the clarity of the model. Non-determinism in a system, for example from communication delays in a network, is also be directly handled in the model. This makes Modbat particularly suitable to test state-based systems with potential non-determinism, such as networked applications [2], [3]. A model can easily be visualized by generating a graph representation using graphviz [4] for a quick overview of all states and transitions. Component-based systems can be modeled using multiple state machines, which are executed in an interleaving way to simulate (stepwise) parallel actions on components. Finally, Modbat provides debugging support on multiple levels by writing error traces into a log and also offering test replay and off-line code generation.

This paper is organized as follows: Section II gives an overview of Modbat. Sections III and IV outline our two demonstrations, on the Java collection classes and the API of a SAT solver (written in C). Section V presents related work.

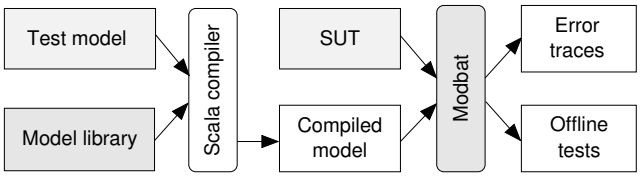

Figure 1. Architecture of and workflow of Modbat.

Section VI discusses Modbat's potential impact on software testing and concludes.

II. Modbat's Architecture and Syntax

Modbat models are written in an internal (embedded) DSL [5] using Scala [6] as host language. We chose Scala because of its flexible syntax [7] and because it runs on the Java VM, which makes it easy to test programs written in Java, Scala, C, and other languages [2]. The DSL mixes deep embedding [5], which defines its own data structure for the abstract syntax tree of the state machine of the model, with shallow embedding for model variables and test actions. This makes the language very concise for the state machine and at the same time allows accessing the full Scala and Java libraries from the model.

On an architectural level, the workflow consists of two steps (see Figure 1): First, the model is compiled against the model library provided by Modbat. The model library defines the DSL in terms of type conversions and custom operators for the deeply embedded part of the DSL, combined with code that can access Modbat's API in test actions, which are written as Scala or Java code. Second, tests are generated. At run-time Modbat loads the model and explores it using a random search, executing the SUT in tandem. The sequence of transitions executed between the initial and final model states constitutes a test run. After each test run, the model and the SUT are reset to their initial state. Upon failure, Modbat emits the error trace showing the execution history.

A. Example: Java's `ArrayList`

Figure 2 shows the syntax of Modbat's DSL on a partial model of Java's `ArrayList`. Each Modbat model extends `modbat.dsl.Model`, which defines the DSL. Variables

```
import modbat.dsl._

class SimpleListModel extends Model {
  val SUT = new java.util.ArrayList[Integer]()
  var n = 0

  def add {
    val ret = SUT.add(new Integer(choose(0, 10)))
    assert (ret)
    n += 1
  }

  def remove {
    require(n > 0)
    SUT.remove(choose(0, n))
    n -= 1
  }

  def bounds {
    choose(
      { () => SUT.remove(-1) },
      { () => SUT.remove(n) }
    )
  }
  "main" -> "main" := add weight 10
  "main" -> "main" := remove
  "main" -> "main" := { assert (SUT.size == n) }
  "main" -> "main" := { SUT.clear; n = 0 }
  "main" -> "main" := bounds throws("IndexOutOfBoundsException")
}
```

Figure 2. Simple list model.

include a reference to the SUT and model variables to track the model's view of the SUT to verify its results. In this case the model only keeps track of the expected size of the list. The model code can also define a number of functions (in the middle) that can be referenced from the declared transitions (at the bottom). This example uses only one model state, *main*, because most actions are available in any state.[1] To use random data, the model calls Modbat's `choose` functions, which return a random number and a random element in a list of functions, respectively.

The example shows that test code can be kept in a separate function or be directly written as an anonymous function. We can also modify the *weight* of a transition function; by default its value is 1. In this example, we emphasize adding elements so calls to `clear` do not repeatedly clear the list before several elements can be added. Finally, we can easily declare that an exception *must* occur in a given transition. In this case, calls to `remove` with index -1 or n access an entry outside the list, so the SUT has to throw a corresponding exception.

B. Key Features of Modbat

Modbat's light-weight DSL has been inspired by a preprocessor to ModelJUnit [8]. Compared to similar tools, it is more concise and expressive for models that are based on transition systems [2], especially for non-deterministic actions like nonblocking network input/output, where the result of an operation depends not only on inputs but also on the physical state of the network [3]. As models are Scala classes, they inherit all variables, functions, and transitions in a natural way, which makes Modbat ideal for testing libraries implementing several related data structures or protocols. Furthermore, the current version of Modbat introduces *observer state machines,* similar to abstract state machines in Spec Explorer [9].

In this demonstration, we take advantage of being able to use multiple models in parallel. Unlike in other tools [9], the number of parallel models does not have to be fixed a priori. Instead, models are instantiated dynamically with function `launch`, which initializes a new (possibly parametrized) model. Newly launched models become active at the end of the current transition.

[1] It is also possible to use multiple states in the model, such as *empty* and *nonempty,* and define transitions with pre- and postconditions.

Table I. ITERATOR METHODS.

Method	Description
`hasNext`	true if forward iterator has more elements
`next`	returns the next element and advances the cursor
`remove`	removes the element that was returned (optional method)

Table II. ADDITIONAL LIST ITERATOR METHODS.

Method	Description
`add e`	inserts the specified element (optional method)
`hasPrevious`	true if reverse iterator has more elements
`nextIndex`	returns the index of the next element
`previous`	return the previous element, moves the cursor backwards
`previousIndex`	returns the index of the previous element
`set e`	replaces the element that was returned (optional method)

III. SCENARIO 1: JAVA ITERATORS

The Java library contains *collections,* data structures for data types such as lists, sets, and maps [10]. Iterators provide a way to access elements of a collection one by one.

A. Semantics of the Iterator API

Iterators can be instantiated on an underlying collection through methods `iterator` and `listIterator`. The former provides a simple forward iterator (see Table I), while the latter provides a bidirectional iterator (see Table II) [10].

Java iterators do not allow a concurrent modification of the underlying collection while iterating on it. Any modification of the underlying collection *invalidates* all previously created iterators on it. Invalid iterators produce an undefined result for calls to `hasNext` and similar methods,[2] and throw a `ConcurrentModificationException` if attempts are made to access or modify data through them. Through experiments we confirmed that this exception is usually only thrown upon a *successful* modification of the underlying collection.

While directly modifying the collection invalidates all of its iterators, it is possible to modify data, if the iterator itself provides a set of optional methods (`add`, `set`, and `remove`). Such modifications are intricately linked with iteration: `remove` and `set` both require a preceding call to either `next` or `previous`. Furthermore, calls to `remove` or `add` require another iterator step before `remove` or `set` can be called again. We discuss this property in more depth below.

B. Model of the Iterator API

Our list model closely mirrors Java's collections but uses simpler data structures, to ensure correctness. We use random data as items to be added, and also add a function to validate internal model invariants. Modbat's support for inheritance is very useful here because `ArrayList` implements a strict subset of all operations in `LinkedList`.[3] Our generic list

[2] This was confirmed by Oracle as a response to a bug report filed by us: http://bugs.java.com/bugdatabase/view_bug.do?bug_id=JDK-8129758.

[3] Some operations are not provided by `ArrayList` because they cannot be implemented efficiently on arrays.

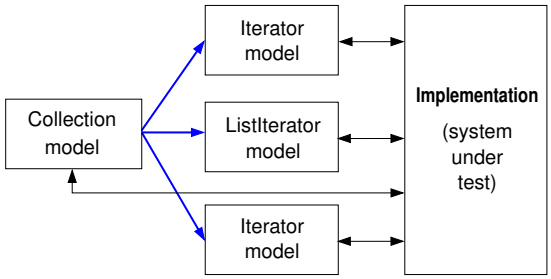

Figure 3. Orchestrating collection and iterator models.

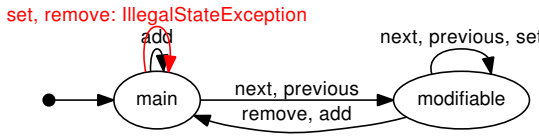

Figure 4. Model of the bidirectional iterator showing the applicability of `set` and `remove` before and after calls to other key methods.

model uses the following data structures: `testData` (the SUT); `data`, a fixed-size array that models the list contents; `n`, which counts the number of elements; and `version`, which counts the number of changes to the collection. We use preconditions to distinguish between cases where a method can be used successfully, and cases where we expect an exception to be thrown. Modbat's direct support for exceptions in transitions allows us to express these features succinctly [2].

Iterators and list iterator models are instantiated by transitions in the list model that launch a new child model instance, and link it to the resulting iterator that is obtained from the SUT. Each model may affect the collection and/or an iterator (see Figure 3). The iterator models remember the `version` of the list so the occurrence or absence of a `ConcurrentModificationException` can be modeled based on whether the version counts of the collection and of the iterator match. Furthermore, we use a state "modifiable" to model whether calls to `set` and `remove` are permitted; these calls require a previous call to `next` or `previous`, without any other modification in between (see Figure 4).

Other requirements are captured using preconditions and postconditions. We choose to model valid and invalid usage contexts using mutually exclusive preconditions, and verify the correct result using exception declarations and postconditions (see Figure 5; details can be found online [11]).

C. Defect Found in Java 1.8

When running the test model against Java's list implementation, some tests fail on `ArrayList`: After a failed `remove(-1)`, the Java library marks the list as modified; subsequent calls to `next` throw a `ConcurrentModificationException` (see Figure 6). Other types of failed modifications (such as `remove` with $n \geq 0$) do *not* mark the list as modified. All other data structures behave consistently in all cases. Modbat finds the problem quickly; Oracle has confirmed the issue as a defect.[4]

[4]https://bugs.openjdk.java.net/browse/JDK-8114832

```
def valid = (version == dataModel.version)
def next { // simplified for brevity
  require (valid)
  require (pos < dataModel.n-1)
  val res = SUTit.next
  pos += 1
  assert (dataModel.data(pos) == res)
}
```

Figure 5. Transition modeling valid uses of `next`.

```
ArrayList<Integer> list = new ArrayList<Integer>;
Iterator<Integer> it = list.iterator();
try {
    list.remove(-1);            // attempt removal
} catch (IndexOutOfBoundsException) {}   // fails
it.next();     // expected: NoSuchElementException
    // but throws ConcurrentModificationException
```

Figure 6. Error trace for Java's `ArrayList`. We obtain an iterator on an empty list, and then try to remove a non-existent element at index -1. This operation does not modify the list, but the next operation throws a `ConcurrentModificationException`, which is wrong. The trace shown here is minimal, but the issue is more serious for non-empty lists, because the spurious exception prevents further access to data.

IV. Scenario 2: SAT Solver

A. SAT Solvers in Verification

SAT solvers are tools for deciding the satisfiability problem of propositional logic. Formulas of propositional logic consist of atomic variables $x, y, \ldots$ defined over the Boolean domain, logical connectives like negation, conjunction and disjunction with standard semantics, as well as parentheses necessary to structure a formula. For example, $(x \vee \neg y) \wedge (\neg x \vee y)$ is true if both variables have the same value. A SAT solver tries to find an assignment for the variables of a formula such that the formula is true under this assignment. By being the prototypical problem for the complexity class NP, SAT offers a powerful framework for encoding and solving many problems stemming from verification, artificial intelligence, etc. [12]. Often SAT solvers are not used only once in a verification problem but in an incremental manner. Here a satisfiable formula is enriched with additional constraints until it becomes either unsat or the considered problem is found to be satisfiable. Usually formulas are represented in conjunctive normal form (CNF), i. e., as conjunction of clauses. A clause is a disjunction of literals and a literal is either a variable or its negation. Constraints are given in form of additional clauses.

For developing a competitive SAT solver that is able to handle real-world formula instances, pruning techniques are essential. These techniques have to be carefully integrated into the incremental solving process in order to preserve the correctness of the solver. Therefore, modern SAT solvers are very complex pieces of software, and as it has been shown they are not resistant to errors. Especially when SAT solvers are part of the verification process, however, it is not acceptable that a SAT solver is faulty, because otherwise the whole effort spent on the verification process is useless. Because of their complex structure, competitive SAT solvers are not amenable to full verification. Therefore, other techniques have to be applied to ensure correctness and trust in SAT solvers.

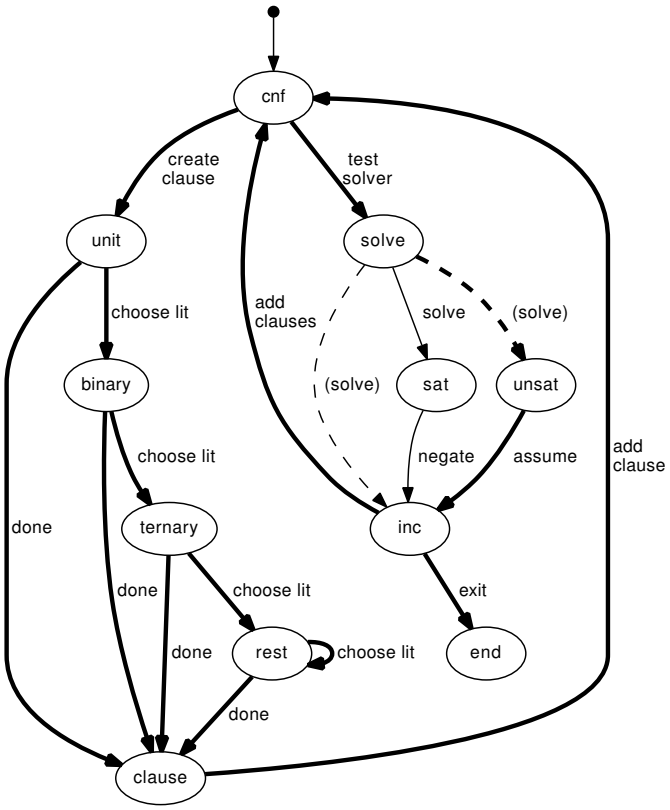

Figure 7. IPASIR model for incremental SAT solvers; coverage after 5 tests.

B. Model-based Testing of SAT Solvers

In previous work, we have presented model-based testing for SAT solvers [13]. To this end, we considered a data model describing expressive random formulas as input as well as a model describing the usage of the API. We presented specific solutions for the SAT solver Lingeling [14], which has always ranked highly in the SAT solver competition because of its sophisticated pruning techniques. In various experiments we showed the power of model-based testing for the SAT solver Lingeling [13]. Those models are very specific to Lingeling, which provides an extensive API and many options, and hence cannot be used for other SAT solvers directly. At that time no standards for incremental SAT solver APIs existed.

This year the SAT competition offers a special track for incremental solving, acknowledging its practical relevance. In this track, submissions include incremental solvers as well as applications which use incremental SAT solvers. To this end, a standardized interface called IPASIR has been specified which the participating SAT solvers have to implement and which can be used by the applications [15]. We contribute to the incremental track of the SAT solver competition by providing a model-based tester for incremental SAT solvers.

We implemented this model-based tester with Modbat by specifying the model shown in Figure 7. The transitions between the states trigger the call of various solver API functions specified by the IPASIR interface. First, the solver is initialized, and then the input formula is generated (left hand side). The input formula consists of unit clauses (clauses of size one), binary clauses, ternary clauses and clauses of arbitrary size. It is important to ensure that the formula contains clauses of size smaller than four because these clauses are often handled in a different manner. The literals and their polarity (negated/not negated) are randomly selected and given to the solver under test. After enough clauses have been generated, the solving function is called. As formulas are generated at random, the outcome of the SAT solver cannot be predicted by the model. The dashed transitions from state solve indicate alternative outcomes (unsatisfiable formulas or a time-out) overriding the default successor state sat, with unsat and inc, respectively, using `nextIf` statements [2] that specify pairs of predicates and successor states. If we use option `--dotify-coverage`, Modbat indicates that the first five tests generate only unsatisfiable formulas (see Figure 7).

The new model (Fig. 7) can be used for any solver implementing the IPASIR interface. Therefore, it can be easily integrated in the solver development process. For the competition we suggest to measure the time a solver takes to finish a given number of tests generated from a fixed random seed.

In the provided demo, we test SAT solver PicoSAT [16]. We demonstrate the tests on the original version 961 and a modified version, where we introduced a small bug in a pruning technique called *failed literal probing* (in the faulty version, a literal is not negated). The bug causes the program to crash sporadically. With Modbat this bug can be found quickly. Note that Picosat can be easily exchanged against any incremental SAT solver implementing the IPASIR interface.

V. Related Work

A variety of model-based testing tools with different features and characteristics have been proposed up to now. In a systematic review, Shafique and Labiche [17] identified a total of 46 tools and 2 APIs providing support for model-based testing. Micskei maintains an online overview of model-based testing tools [18]; currently (according to the information last modified in July 2014) the list contains 20 academic, commercial and open source tools plus 13 tools that are not developed anymore. Several tools have features and characteristics that are similar to those of Modbat. To highlight and explain the differences between Modbat and related tools, we selected three widely-known tools for comparison: ModelJUnit [1], ScalaCheck [19] and Spec Explorer [9]. We consider these tools to be representative examples as we have practical experience from applying them in previous projects and since some of their features were a source of inspiration when implementing Modbat. References to further tools are included in the discussion of Modbat's specific features below.

The underlying modeling approach has a big impact on the implementation and features of the tools. Like in ModelJUnit [1], Modbat's models are based on extended finite-state machines (EFSMs). Yet there is a wide range of different modeling approaches used by different tools, for example, RT-Tester [20] and MoMuT::UML [21] use UML, T-VEC [22] uses models from Simulink, LOTOS, timed automata, etc. Furthermore, TTCN-3 is a popular testing language to model communication systems and protocols, and it is supported by various tools [23].

Compared to other notations, EFSMs are simple but expressive, and can be readily integrated with existing test code

Table III. CHARACTERISTICS OF RELATED TOOLS.

		Modbat [2]	ModelJUnit [1]	ScalaCheck [19]	Spec Explorer [9]
Model specification	Scope	input + output	input + output	input + output	input + output
	Characteristics	untimed	untimed	untimed	untimed
		non-deterministic	deterministic	deterministic	non-deterministic
		discrete	discrete	discrete	discrete
	Paradigm	transition-based	transition-based	generator-based	state-based + history-based
	Interface	internal DSL	API	API	external DSL
Test generation	Test selection criteria	random + stochastic	random + stochastic	random + stochastic	structural model coverage + data coverage
	Technology	random + search-based	random + search-based	random + search-based	model checking
Test execution	Online/offline	both	online	online	both
License		base version open source, extended version proprietary	open source	open source	proprietary

because the DSL or API is hosted by a widely used execution platform (in many cases, the Java VM) [7]. This advantage makes such tools ideally suited to test software directly, without any external test harness. Other tools like OSMO and NModel also use state machines and take a similar approach as ModelJUnit; the structure of the model is defined via annotated methods [24], [25]. In general, these tools are related to Modbat as they model both inputs and outputs, and have a notation of discrete transitions without timing constraints. However, models differ in whether the SUT (and its output) is considered to be strictly deterministic, and whether the model is centered on actions, random choices (specified by generators), or states and transition histories. A key feature distinguishing Modbat from other tools is its ability to deal with non-determinism.

Model-based testing tools can be classified according to whether they provide an API based on a well-known programming language (e. g., Java, C#) or a DSL (which further can be classified into an external or internal DSL) for describing SUT models. An advantage of API-based approach is that practitioners can use their familiar languages. Yet a DSL provides an easy to understand, user-friendly interface for testers writing and maintaining models as they making models potentially less complex [26]. Describing SUT models with Modbat's DSL on top of Scala combines the advantages of both approaches. Test selection criteria and search techniques for space exploration of the tools differ as well; furthermore, some tools generate off-line test code, which can be run without executing the state space exploration again.

Table III shows Modbat in comparison with the three selected tool examples. The comparison mainly follows the lines of Utting et al.'s taxonomy of model-based testing approaches [27], additionally considering the model specification interface and the license. ModelJUnit [1] inspired the initial version of Modbat [8] and takes a similar approach but is API-based. With several new versions of Modbat, many more features (test case generation for offline testing, debugging support, support for non-determinism, etc.) have been added. We consider ModelJUnit as a representative example for similar tools such as OSMO and NModel [25], [24]; the latter has been superseded by Spec Explorer. ScalaCheck [19] is a tool for property-based testing [28], primarily designed to generate complex data with constraints, but it can also be used to model transition systems and supports stateful testing. Like Modbat, it is based on Scala and shares some of its features [19]. Spec Explorer [9] is another model-based testing tool that provides a DSL to model state machines, although it also enables mainstream programming languages (e. g., C#) as input notation [29]. Spec Explorer is a commercial tool developed by Microsoft. It is based on the Windows/.NET platform, whereas Modbat runs on Java VMs available for many of the major platforms. Other differences to Spec Explorer are Modbat's dynamic instantiation and the support for exception handling on the model level.

VI. DISCUSSION AND CONCLUSION

ModelJUnit [1] helped to bring model-based testing to early adopters of test case generation technology. Modbat has been inspired by ModelJUnit [8] and has been created to simplify some of the modeling tasks, such as specifying transitions with their preconditions and checking the occurrence of exceptions in actions [2]. It has been successfully used to model complex systems like SAT solvers [2], where it replaced a custom test generator written in C [13], and the Java network library, including non-deterministic actions like non-blocking input/output [3]. This demonstration shows Modbat on Java's iterators, where we found a new previously unknown defect, and on a new API for incremental SAT solving. Modbat's flexible DSL makes it possible to express both models succinctly and clearly. In our opinion, this facilitates focusing on the semantics of a system, which reduces the model development time and the occurrence of defects in the model. We think that Modbat contributes to making model-based testing more applicable to complex software, and we hope that with an open source release of Modbat, model-based testing will become more widespread.

Even with an elegant and expressive modeling platform, writing a model that includes an output oracle requires an in-depth understanding of the system. Good model design is not always straightforward. In our experience, cognitive bias has sometimes prevented us from modeling the full state space in an initial version of the model [30]. We plan to add graphical tool support, and more ways to visualize model traces and their code coverage in the future, to mitigate this problem.

Modbat is available for download [11]. Material for this demonstration can be found online [31].

ACKNOWLEDGEMENTS

Part of the work was supported by the Japanese Society for the Promotion of Science (*kaken-hi* grants 23240003 and 26280019), by the Austrian Science Fund (FWF) through the national research network RiSE (S11408-N23), by the Austrian Ministry for Transport, Innovation and Technology, the Federal Ministry of Science, Research and Economy, and the Province of Upper Austria in the frame of the COMET center SCCH.

REFERENCES

[1] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. San Francisco, USA: Morgan Kaufmann Publishers, Inc., 2006.

[2] C. Artho, A. Biere, M. Hagiya, E. Platon, M. Seidl, Y. Tanabe, and M. Yamamoto, "Modbat: A model-based API tester for event-driven systems," in *Proc. 9th Haifa Verification Conference (HVC 2013)*, ser. LNCS, vol. 8244. Haifa, Israel: Springer, 2013, pp. 112–128.

[3] C. Artho, M. Hagiya, R. Potter, Y. Tanabe, F. Weitl, and M. Yamamoto, "Software model checking for distributed systems with selector-based, non-blocking communication," in *Proc. 28th Int. Conf. on Automated Software Engineering (ASE 2013)*. Palo Alto, USA: IEEE Computer Society, 2013, pp. 169–179.

[4] E. Gansner and S. North, "An open graph visualization system and its applications to software engineering," *Software—Practice and Experience*, vol. 30, no. 11, pp. 1203–1233, 2000.

[5] D. Wampler and A. Payne, *Programming Scala*, ser. O'Reilly Series. O'Reilly Media, 2009.

[6] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala: A Comprehensive Step-by-step Guide*, 2nd ed. USA: Artima Inc., 2010.

[7] C. Artho, K. Havelund, R. Kumar, and Y. Yamagata, "Domain-specific languages with scala," in *Proc. 17th Int. Conf. on Formal Engineering Methods (ICFEM 2015)*, ser. LNCS, 2015, to appear.

[8] C. Artho, "Separation of transitions, actions, and exceptions in model-based testing," *Post-proceedings of 12th Int. Conf. on Computer Aided Systems Theory (Eurocast 2009)*, vol. 5717, pp. 279–286, 2009.

[9] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson, "Model-based testing of object-oriented reactive systems with Spec Explorer," in *Formal Methods and Testing 2008*, ser. LNCS, vol. 4949. Springer, 2008, pp. 39–76.

[10] *Java Platform SE 8*, Oracle, Santa Clara, USA, 2015, `http://docs.oracle.com/javase/8/docs/api/`.

[11] C. Artho, "Modbat," 2015, https://staff.aist.go.jp/c.artho/modbat/.

[12] A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds., *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press, 2009.

[13] C. Artho, A. Biere, and M. Seidl, "Model-based testing for verification backends," in *Proc. 7th Int. Conf. on Tests & Proofs (TAP 2013)*, ser. LNCS. Springer, 2013, pp. 39–55.

[14] A. Biere, "Yet another local search solver and Lingeling and friends entering the SAT competition 2014," *SAT Competition 2014*, vol. 2014, p. 2, 2014.

[15] T. Balyo, , C. Sinz, M. Iser, and A. Biere, "SAT-race 2015," 2015, http://baldur.iti.kit.edu/sat-race-2015/.

[16] A. Biere, "PicoSAT," 2015, http://fmv.jku.at/picosat/.

[17] M. Shafique and Y. Labiche, "A systematic review of state-based test tools," *International Journal on Software Tools for Technology Transfer*, vol. 17, no. 1, pp. 59–76, 2013.

[18] Z. MICSKEI, *Model-based testing (MBT)*, 2014, http://mit.bme.hu/~micskeiz/pages/modelbased_testing.html.

[19] R. Nils, "ScalaCheck, a powerful tool for automatic unit testing," 2013, https://github.com/rickynils/scalacheck.

[20] J. Peleska, "Industrial-strength model-based testing - state of the art and current challenges," in *Proc. Eighth Workshop on Model-Based Testing*, 2013, pp. 3–28.

[21] W. Krenn, R. Schlick, S. Tiran, B. Aichernig, E. Jobstl, and H. Brandl, "Momut::UML model-based mutation testing for UML," in *Proc. Software Testing, Verification and Validation (ICST)*, 2015, pp. 1–8.

[22] "T-VEC," `http://www.t-vec.com`.

[23] C. Willcock, T. Deiss, S. Tobies, S. Keil, F. Engler, S. Schulz, and A. Wiles, *An Introduction to TTCN-3*, 2nd ed. Wiley, 2011.

[24] J. Jacky, M. Veanes, C. Campbell, and W. Schulte, *Model-Based Software Testing and Analysis with C#*, 1st ed. Cambridge University Press, 2007.

[25] T. Kanstrén and O. Puolitaival, "Using built-in domain-specific modeling support to guide model-based test generation," in *Proc. 7th Workshop on Model-Based Testing (MBT 2012)*, ser. EPTCS, vol. 80, 2012, pp. 58–72.

[26] S. Sobernig, P. Gaubatz, M. Strembeck, and U. Zdun, "Comparing complexity of API designs: an exploratory experiment on DSL-based framework integration," *International Journal on Software Tools for Technology Transfer*, vol. 47, no. 3, pp. 157–166, 2011.

[27] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," *Softw. Test. Verif. Reliab.*, vol. 22, no. 5, pp. 297–312, Aug. 2012.

[28] K. Claessen and J. Hughes, "Quickcheck: a lightweight tool for random testing of Haskell programs," *SIGPLAN Not.*, vol. 46, no. 4, pp. 53–64, 2011.

[29] W. Grieskamp, "Multi-paradigmatic model-based testing," in *Formal Approaches to Software Testing and Runtime Verification*. Springer, 2006, pp. 1–19.

[30] C. Artho, K. Hayamizu, R. Ramler, and Y. Yamagata, "With an open mind: How to write good models," in *Proc. 2nd Int. Workshop on Formal Techniques for Safety-Critical Systems*, ser. CCIS, no. 419. Queenstown, New Zealand: Springer, 2013, pp. 3–18.

[31] C. Artho, M. Seidl, Q. Gros, E. Choi, T. Kitamura, A. Mori, R. Ramler, and Y. Yamagata, "Modbat tool demonstration," 2015, https://staff.aist.go.jp/c.artho/modbat/tooldemo/.