Modbat: A Model-based API Tester for Event-driven Systems

Cyrille Artho[1], Armin Biere[2], Masami Hagiya[3], Eric Platon[4], Martina Seidl[2], Yoshinori Tanabe[5], and Mitsuharu Yamamoto[6]

[1] Nat. Ins. of Advanced Industrial Science and Technology (AIST), Amagasaki, Japan
[2] Johannes Kepler University, Linz, Austria
[3] The University of Tokyo, Tokyo, Japan
[4] KVH, Inc., Tokyo, Japan
[5] National Institute of Informatics (NII), Tokyo, Japan
[6] Chiba University, Chiba, Japan

Abstract. Model-based testing derives test executions from an abstract model that describes the system behavior. However, existing approaches are not tailored to event-driven or input/output-driven systems. In particular, there is a need to support non-blocking I/O operations, or operations throwing exceptions when communication is disrupted.
Our new tool "Modbat" is specialized for testing systems where these issues are common. Modbat uses extended finite-state machines to model system behavior. Unlike most existing tools, Modbat offers a domain-specific language that supports state machines and exceptions as first-class constructs. Our model notation also handles non-determinism in the system under test, and supports alternative continuations of test cases depending on the outcome of non-deterministic operations.
These features allow us to model a number of interesting libraries succinctly. Our experiments show the flexibility of Modbat and how language support for model features benefits their correct use.

Keywords: Software testing, model-based testing, test case derivation

1 Introduction

Software testing executes parts of a system under test (SUT) [20]. A series of inputs is fed to the SUT, which responds with a series of outputs. In *model-based testing*, test cases are derived from an abstract model rather than implemented directly as code. This approach has several advantages: A high-level model is easier to develop and understand than program code; and partially specified behaviors give rise to many possible combinations, from which many concrete test cases can be derived.

Many existing test generation tools are designed to generate test *data* to test a given function or method [9,17,21]. Our work covers test *actions* instead, spanning sequences of multiple function calls.

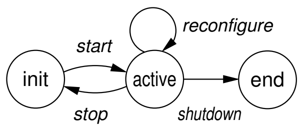

Fig. 1. A finite-state machine modeling component behavior.

Existing tools to generate test actions [9,15,16,21,28,29] define a programming interface against which the model is coded; the programmer defines classes and functions that implement the semantics of the model. It was shown that this introduces a layer of abstraction in the model that makes it difficult to clearly express the behavior of event-driven systems, which include databases, file systems, and cloud computing middleware [2]. Such systems also often depend on possibly unreliable hardware or communication links, which introduce possible delays and failures.

Our tool *Modbat* provides a domain-specific language to model test executions in such systems succinctly. In Modbat, system behavior is described using extended finite-state machines (see Figure 1). They provide a graphical, intuitive base of the model, which can be refined using a domain-specific language provided by Modbat. For example, the user can add additional preconditions about when given events are permissible. Results can be checked using assertions, or stored in model variables to be used in subsequent calls.

1.1 Usage of Modbat

Modbat strives to simplify test modeling by offering a flexible modeling platform. A tester uses Modbat as follows:

1. The tester defines a model as an extended finite-state machine. The model is compiled against a library provided by Modbat. For example, a transition from Figure 1 could be written as
   `"init" -> "active" := { c = new Component; c.start }`.
2. The tester runs Modbat against the compiled model. Modbat explores the possibilities defined by the model (the state space) using a random search, executing the SUT in tandem. After each completed test, the model and the SUT are reset to their initial state.
3. Modbat either executes a predefined number of tests, or it runs until a failure is found. A failure is detected when a test run violates a property. When a failure is found, Modbat writes an error trace to a file, giving the necessary information to analyze the error. For debugging, a failed test can be replayed.

1.2 Outline

This paper is organized as follows: Section 2 gives the necessary background and describes related work. Section 3 presents our modeling notation, and Section 4 describes our tool Modbat. Section 5 covers projects carried out with Modbat. Section 6 concludes and outlines future work.

2 Background

2.1 Terminology

A system under test (SUT) exposes its functionality to other software via an application programming interface (API). APIs may be organized in libraries.

Testing executes parts of the SUT [20]. A *test trace,* implemented as a *test case,* is a series of function calls (method invocations in object-oriented languages). A *test run* is the execution of a test case at run-time. The *test harness* serves to set up and manage test execution. A *test suite* is a set of test cases.

Our *test model* is based on extended finite state machines (EFSM) [8]. Formally, an EFSM is defined as a 7-tuple $M = (I, O, S, D, F, U, T)$ where

- S is a set of states, I is a set of input symbols, O is a set of output symbols;
- D is an n-dimensional vector space $D_1 \times \ldots \times D_n$,
- F is a set of *enabling functions* $f_i : D \rightarrow \{0, 1\}$,
- U is a set of *update functions* $u_i : D \rightarrow D$, and
- T is a transition relation $T : S \times F \times I \rightarrow S \times U \times O$ [8].

In an EFSM, D models the internal state of the model; the enabling functions describe when transitions are enabled; and update functions change the internal state of the model based on the outcome of a function call to the SUT.

In the formal definition, D is a vector space of fixed dimension. In our tool (see Section 3), we allow dynamic memory allocation, but for the purpose of describing the tool behavior, this difference is not important.

A test model usually has a large degree of uncertainty so that it can describe many possible test executions. *Test derivation* (test generation) describes the action of deriving individual concrete test cases from the abstract test model.

Techniques that rely solely on the specification of a system are called *black-box* techniques, while approaches that consider the implementation during analysis are known as *white-box* techniques. In our case, the specification is sufficient to write a test model; hence we consider our approach black-box.

2.2 Online vs. Offline Testing

Test derivation techniques can be divided into online testing and offline testing [28]. In online testing, the test generation (derivation) tool connects directly to the SUT. Each test is directly executed while it is generated (see Figure 2).

In offline test generation (see Figure 3), the test derivation tool does not execute the SUT directly. Instead, it generates an intermediate representation of test cases that is turned into executable tests later [28]. Existing literature also covers generation of manually deployable tests, which is elided in this paper [28].

Online testing is more efficient than offline testing, as no test code needs to be generated and compiled. A model can be set up to handle non-determinism in the SUT (also see Section 3), and may even dynamically fine-tune the test model based on earlier test executions.

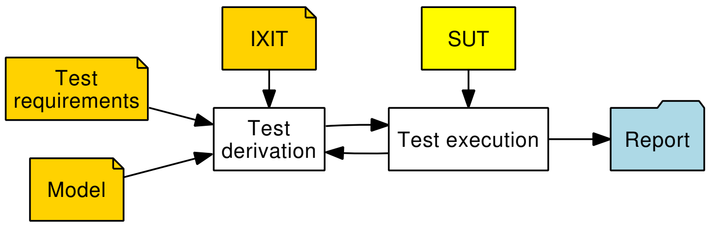

Fig. 2. Online testing. Test requirements include configuration options for the test tool, such as the number of tests to be generated; IXIT refers to *implementation extra information,* which is information needed to derive executable test cases [28]. Such information includes system configuration (such as library locations).

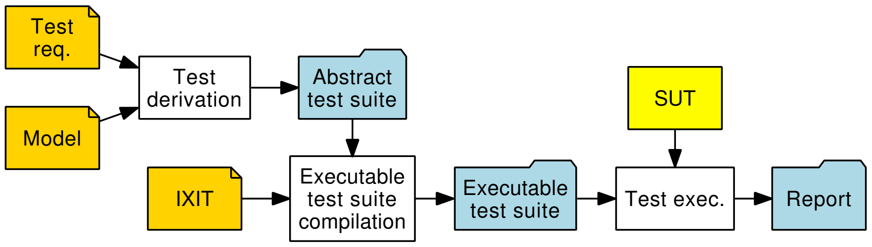

Fig. 3. Offline testing: Abstract tests are turned into executable tests later.

Offline testing has the advantage that the test execution platform does not depend on the test derivation tool. The execution platform is therefore simpler and has a higher chance of not introducing unwanted behavior. Offline testing is also applicable when test cases have to be executed on a platform that is not supported by the test derivation tool. Modbat supports both online and offline testing, so it can handle a large set of use cases and platforms.

2.3 Related Work

Unit testing experienced a widespread rise in software development in the late 1990s [6,7,19]. In particular, JUnit is widely used for Java programs [19]. Modbat supports offline code generation for JUnit, allowing it to complement legacy test suites with automatically generated tests.

Many model-based testing tools allow a model to be defined against a programming interface (API). Such tools include QuickCheck [9] and ScalaCheck [21], which generate data based on predicate constraints; that data is then used to execute a given function in the SUT. One of the earliest practical model-based test tools that generates test sequences was ModelJUnit [28], which is also based on extended finite state machines. In ModelJUnit, a lot of the structure of the EFSM is implemented by the user (also see Section 3). A tool that motivated Modbat was actually implemented as a preprocessor to ModelJUnit [2]. Osmo and NModel also use state machines and take a similar approach as ModelJUnit;

the structure of the model is defined via annotated methods [15,16]. NModel and its successor SpecExplorer [29] also distinguish themselves by splitting the model into a scenario model (generating the tests) and a contract model (verifying the outcome), while most other tools combine both aspects in one model.

Model-free test case generation techniques require no user-defined model. Randomized testing is such a white-box technique, which executes the SUT without any specification. Instead, it explores possible test actions and tries to execute as many aspects of the SUT as possible. Such testing may be primarily randomized [10,24] or more systematic and coverage-driven [11].

Concolic testing tools also optimize test coverage by keeping track of *(symbolic)* constraints that describe if a code branch can be taken; these constrains are then refined with current values during *concrete* test execution [13,27].

Model-free approaches essentially reverse engineer the models from the code. Full automation comes at the expense of having no output oracle is available, limiting detectable failures to executions that result in a fatal outcome (typically a crash or unhandled exception).

3 Modeling Notation

We address the problems listed in Section 1 by the following design decisions:

1. The underlying model is an extended finite state machine (EFSM), which is well understood by the formal methods community and developers alike. Transitions in the EFSM are directly linked to program code.
2. The model is not expressed as a program, but in a domain-specific language (DSL). Our DSL supports non-determinism both in the specification (to simulate faults or non-deterministic events) and in handling the resulting SUT behavior (to handle faults or exceptions). It is, to our knowledge, the only tool both supporting non-determinism and exceptions as primary constructs.

Figure 4 shows an example. Class `SimpleCounter` offers two methods that can increase its counter value (initially 0). The first method has the added property that a successful outcome depend on a flag (initially *true*). If the flag is set to *false* by calling `toggleSwitch`, then calling `inc` does not change the counter value. A corresponding test model calls the given methods in random order until value 2 is reached. In doing so, it does not take the side-effect of `toggleSwitch` into account. Modbat finds sequences of actions in the model that result in violating the assertion in the model, such as `toggleSwitch`, `inc`, `inc`, `assert`.

Note that a labeled state in an EFSM does not reflect the full model state; in our example, states correspond to the counter value but do not include the state of the switch in the counter (see Figure 5). This design is deliberate: The choice of a more abstract model state keeps the model size small. Model variables are used instead to define details matching the precise system state [28].

As a consequence of this design approach, different paths resulting in the same model state may correspond to different system states. In other words, model state $\langle s, D \rangle$ and SUT state do not necessarily match for two traces where

```
public class SimpleCounter {
  int count = 0;
  boolean flag = true;

  public void toggleSwitch() {
    flag = !flag;
  }
  public void inc() {
    if (flag) {
      count += 1;
    }
  }
  public void inc2() {
    count += 2;
  }
  public int value() {
    return count;
  }
}
```

```
class CounterModel extends Model {
  var counter = new SimpleCounter()
  // transitions
  def instance() = { new MBT (
    "zero" -> "zero" := {
      counter.toggleSwitch
    },
    "zero" -> "one" := {
      counter.inc
    },
    "one" -> "two" := {
      counter.inc
    },
    "zero" -> "two" := {
      counter.inc2
    },
    "two" -> "end" := {
      assert (counter.value == 2)
})}}
```

Fig. 4. Example system (left) and Modbat model (right).

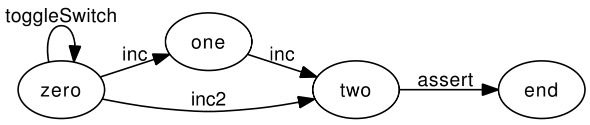

Fig. 5. Visualization of example model.

$s = s'$ due to abstractions in the model. We are not aware of an established best practice for choosing the best level of abstraction, but it is well-known that choosing the right level of abstraction is a challenging task [18].

3.1 ESFM Notation

The notation of our model is motivated by previous work [2], which used a model that is based on graphs expressed in the notation used by Graphviz [12]. We decided to move to a new notation, because the Graphviz format is not extensible and not designed for model-based testing, but for visualization.

Our notation is designed as an embedded DSL [31] on top of the Scala programming language [22]. Transitions are declared with a concise syntax:

`"pre_state" -> "post_state" := { transition_action }`.[7]

Transition actions include calls to the SUT API, and assertions that check return values against given properties. Transition actions can be modified by adding declarations on exceptional or non-deterministic behavior.

[7] The first declared state automatically constitutes the initial state of the model.

Exceptions can be specified succinctly in Modbat. If an exception of a given type is always expected, `throws("ExceptionType" {, "ExcType"})` gives a list of possible exceptions, where one of them *must* occur during the transition. If an exception *may* occur, `catches` can be used, which specifies the target state for the model if the exception occurs:

```
"pre" -> "post" := { action } catches ("Exception" -> "errorState").
```

Related to the formal definition of ESFMs in Section 2, Modbat does not use explicit input and output symbols; input/output functions of Java's libraries are available instead. D is represented by model variables. F is implemented by Modbat itself as a check of the predecessor state, and whether declared preconditions evaluate to *true*. Update functions U are transition actions as shown above; and the transition relation T, implemented by Modbat, takes into account the given successor states, exceptions, and the semantics of `nextIf` as shown below.

Modbat also supports some shorthand definitions, for example to model a set of transitions with the same action and end state, and to model multiple transitions with the same predecessor and successor states but different actions. We believe that automatically combining user-defined annotations for preconditions and exceptions with transitions derived from the EFSM, is a key to making a test derivation tool both expressive and easy to use. Earlier work has shown that managing the model state as user-defined code is tedious and error-prone [2].

3.2 Comparison to API-driven Test Derivation Tools

Existing tools like ModelJUnit [28] or ScalaCheck [21] require user-defined code to manage the model state. To show the benefit that Modbat provides, we compare the necessary code to manage the model state for normal transitions and transitions throwing exceptions. The overhead of user-defined code becomes particularly high when exceptions have to be managed. Figure 6 shows, for different tools, the code for managing the model state, and for ensuring that an exception *always* occurs (for example, when `action` is executed in the wrong system state). Code to manage optional exceptions is similar in ModelJUnit and ScalaCheck; it updates the next state instead of a boolean and contains no assertion.

Compared to other tools, Modbat's notation is more concise (see Table 1). When counting the additional code to handle certain features, we assume a non-trivial EFSM with non-empty preconditions and do not count the action function itself, or lines containing only curly braces. We also count a state transition in Modbat as two lines, as it is usually formatted as such (see Figure 4). We can see that our notation reduces the amount of model code by 75 % for each transition while also eliminating error-prone repetitive code that manages internal information.[8]

Preconditions (which may not contain side-effects) are supported by all tools in similar ways. In Modbat, preconditions are declared using `require`, using the same syntax as in normal Scala code.

[8] Defining additional helper functions in ScalaCheck could partially alleviate the overhead of exception checking, but not of state management.

```
       Modbat                  ModelJUnit                   ScalaCheck
"pre" -> "post" :=   boolean action0Guard() {   case object Transition0
{ action }             return state == 0; }       extends Command {
                     @Action void action0() {   preConditions += (s==...)
                       action();                def run(s: State) = action
                       state = 1; }             def nextState(s:State)=...

       Modbat                  ModelJUnit                   ScalaCheck
{ action }              { boolean ok = false;      { var ok = false
throws("Exception")     try {                      try {
                          action();                  action
                        } catch(Exception e) {     } catch { case e:
                          ok = true; }               Exception => ok=true }
                          assert (ok); }             assert (ok) }
```

Fig. 6. State transitions (top) and exception handling (bottom) in different tools.

Table 1. Lines of additional code needed (per use) to handle certain model features.

Feature	Modbat	ModelJUnit	ScalaCheck
State transition	1	4	4
Precondition	1	1	1
Expected exception	1	5	5
Optional exception	1	4	4

As can be seen, Modbat's embedded DSL supports model constructs efficiently while still leveraging the host language, Scala. Other related tools also use models defined in program code against a given API (see Section 2.3); the coding overhead compared to our DSL-based approach is similar to the cases shown in Table 1.

3.3 Advanced Modeling Features: Non-determinism and Annotations

We consider non-determinism a powerful modeling feature and therefore support multiple operators that deal with different types of non-determinism:

Choose returns a random number in a given range. It is intended to cover relatively simple use cases. We do not model complex data such as supported by generator functions in ScalaCheck [21], because the generators from ScalaCheck can be combined with Modbat as long as replay and offline testing (see below) are not used.

Maybe executes code probabilistically, with default probability of 0.5. This is useful when testing SUT functionality that is expected to fail sometimes (such as input/output). In a normal test setup, failure probability may be too low to be observed. With `maybe`, faults can be simulated (injected) on the model level.

NextIf models alternative outcomes on the SUT side. For instance, non-blocking input/output (I/O) operations return immediately but may be incomplete. Using `nextIf`, the model can account for both possible outcomes of such an operation.

When using non-blocking I/O, a failed or partial I/O operation usually has to be retried later. In Modbat, `nextIf` overrides the normal successor state if its predicate is *true*. The following code models a non-blocking accept call in `ServerSocketChannel`, which is part of package `java.nio` in the standard Java library [23]:

```
"accepting" -> "connected" := {
  connection = ch.accept()
} nextIf ({ () => connection == null} -> "accepting").
```
[9]

The documentation of `accept` states that the non-blocking variant immediately returns a new connection handle if it is successful and `null` otherwise [23]. The normal successor state in the model (*connected*) covers the case where the operation is successful, but `nextIf` defines an alternative to remain in the current model state otherwise.

Using `nextIf` allows exhaustive testing of component-based systems which are often I/O-driven and have to be able to continue after communication problems. It is similar to next-state declarations in modeling languages such as Promela [14] used for the exhaustive verification of algorithms. As Modbat tests implementations, exhaustive verification cannot be attained directly. To observe all possible outcomes, either a large number of test runs is needed, or a platform that can simulate delays, such as Java PathFinder [30]; also see Section 5.

As our modeling language is embedded in Scala, all Scala language features are available. In particular, longer blocks of code can be written as separate functions. Functions may also be *annotated* so they have a special significance outside a test run: Functions annotated with `@init` and `@shutdown` are executed before the first and after the last test run, respectively. In our case studies, we used these annotations to run a server in the background while client tests were being executed, and to shut the server down cleanly at the end. While this task could also be handled externally (e. g., by shell scripts), having a built-in mechanism allows us to refer to model data. Similarly, annotations `@before` and `@after` execute a function before and after *each* test. Such functions set up and tear down auxiliary data structures without complicating the test model itself.

3.4 Other Features

Modbat records any decisions taken by its own random number generator (RNG), so tests can be replayed in full (with a given random seed) or partially (with a subset of the trace of random numbers generated). Unfortunately, replaying currently does not support third-party data generators such as ScalaCheck's.

[9] The parentheses followed by the arrow, `() =>`, are a Scala syntax artifact to encapsulate the condition in an anonymous function, which is then passed to Modbat.

$$\begin{tikzcd}
\text{Test 1} & \bullet \arrow[r] \arrow[d] & \circ \arrow[r] & \circ \arrow[r] & \circ \\
\text{Test 2} & \bullet \arrow[r] \arrow[d] & \circ \arrow[r] & \circ \arrow[r] & \dots \\
\vdots & \vdots & & &
\end{tikzcd}$$

Fig. 7. Master (filled circle)/slave (unfilled circle) RNG usage in Modbat.

Modbat uses random numbers in a tree-like fashion: Instead of one RNG, a *master RNG* and a *slave RNG* are maintained (see Figure 7). The master is advanced by one step at the end of each test run and only used to seed the slave RNG for a new test run. Because the RNG is restored to a new seed before a new test is started, changes in the number of (slave) RNG calls in previous test cases do not affect the new state at the beginning of a new test. This keeps test runs within a test suite consistent across minor modifications of the model (such as adding an extra choice that does not affect the outcome of most test runs). Precomputing master RNG states even allows parallelization of test executions.

Modbat supports multiple ESFMs, which are executed using an interleaving semantics. Modbat starts a given state machine first, which then uses the `launch` function of Modbat to launch a (parametrized) instance of another ESFM.

Modbat supports both online and offline testing, even when random numbers are used. For offline testing, the trace of random numbers is recorded for each transition, so test case minimization can be performed later [32].

Modbat also supports coverage measurement; currently, state and transition coverage are measured. Coverage can be visualized (along with the test model) using Graphviz [12].

4 Implementation Architecture

Modbat's test models are defined in an embedded DSL [31] on top of Scala [22]. Scala was chosen for its extensible syntax (allowing the definition of domain-specific languages on top of Scala) and its compatibility with Java. We use the Scala compiler to parse the test model and compile it against a model library that defines the syntax of our DSL and its operators (see Figure 8).

Modbat supports a variety of configuration options. Test requirements include the total number of tests and the option to abort after a failure. The length of test runs can also be limited by setting a probability to abort a test after each step. Implementation-specific information (IXIT in Figure 2) includes library locations, log file locations, and the option to set a random seed.

Modbat loads the compiled model and tests it against the SUT. Normally, Modbat runs in the normal Java Virtual Machine (JVM), and executes test cases online against the test specification. Failed tests are reported and can optionally

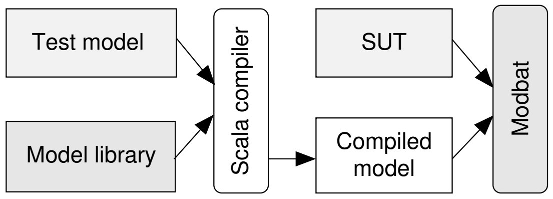

Fig. 8. Implementation architecture (online testing).

be written to a Java or Scala source file as unit tests. Modbat supports JUnit-style test code generation or a stand-alone format that uses its own test harness.

Offline test code is compiled against the test model for later execution. Both the compiled unit tests (defining the test traces) and the test model are used for offline testing, as the latter holds model variables and state transition functions.

5 Usage of Modbat in Software Development

To find defects in Modbat itself, Modbat is tested internally against a number of test models. Tests range from parsing command line arguments to online and offline test derivation. About 200 such tests are used as regression tests during development, to validate Modbat itself. Online test case generation is tested by running Modbat on the compiled test model, and observing the outcome (test coverage reported, number of failures found, contents of error traces). For offline testing, two more stages are added: First, the generated source code is compiled and compared against a previous run; then, the output of the executed offline tests is also verified.

5.1 Testing the SAT Solver Lingeling with Modbat

Our first project illustrates how Modbat is used to test SAT solvers, which are very complex and highly optimized programs. As SAT solvers often serve as reasoning backends in verification frameworks, their correctness is of particular importance. In general, efficiently implemented SAT solvers are too complex to be fully verified themselves, so testing approaches like model-based testing are required to ensure their robustness. Previous work [3] shows how model-based testing compares to standard testing techniques for the SAT solver Lingeling, which is the winner of several major tracks at the SAT solver competition 2013. Besides standard reasoning techniques found in almost all recent SAT solvers, Lingeling additionally implements several preprocessing rules which are very effective, but which also increase the complexity of the solver's code and API.

An earlier testing framework [3] realizes a model-based testing approach where the model is expressed in plain C code. The generation of the test cases based on this model was also manually implemented in C, requiring about 650 lines of code. We formulated the same model, consisting of 12 states with 18 transitions, in Modbat with only 300 lines of code for the same functionality. As Lingeling is written in C, we used the Java Native Access (JNA) framework to make Lingeling's API accessible in Modbat. We did not experience any restrictions concerning the expressiveness of neither Modbat nor Scala used for describing the generation of the input data. Because each test run consists of hundreds of transitions with almost equally many (uncovered) branches, 99.98 % of all generated paths by Modbat were unique after 100,000 tests.

We performed a similar experiment as described in earlier work [3], applying our tests to 373 different defective versions of Lingeling. The faults were randomly seeded into Lingeling's code by removing arbitrary lines or by introducing abort-statements. Each generated test case was then applied to each defective version, with a timeout of 100 seconds per test case (on a given instance of Lingeling). In our setup, 55 % of the defects were found with the model written in C [3] within that timeframe. With Modbat we found 60 % of the defects. When looking at the execution performance, we experienced that the hand-coded model in C is only twice as fast as the generic Modbat framework.

5.2 Java PathFinder

To evaluate Modbat to an input/output (I/O) driven system, we applied it to an implementation of `java.nio`, Java's network library for non-blocking, selector-based I/O. This library is used in Java PathFinder (JPF) [30], which is a software model checker for Java bytecode. In a concurrent SUT, JPF explores the outcomes of all possible thread interleavings by backtracking executions to a previously stored program state, and exploring the other remaining outcomes from that point. By itself, JPF cannot handle network I/O, because backtracking the SUT causes it to be out of sync with its environment [5].

In JPF, a *model library* can provide the missing functionality that JPF does not support. Other work describes the implementation of `java.nio` in JPF and its application to a web server in depth, along with a preliminary evaluation of an earlier model [4]. In this work, we describe our enhancements to the earlier model [4], the new defects that Modbat revealed, and its behavior in JPF in more depth.

JPF implements software model checking by running the SUT in its own virtual machine (VM) that is capable of backtracking the entire SUT state. JPF currently supports only the execution of entire programs; therefore, the test harness also needs to be executed inside JPF, even if it is entirely deterministic. In our case, Modbat is the test harness, and the `java.nio` library is the SUT.

5.3 Evaluation of Models for Non-blocking I/O

Non-blocking I/O is difficult to test: A non-blocking operation returns immediately, but the result may be incomplete. The correct use of non-blocking I/O therefore requires state and buffer management code to ensure completion of an operation. While more difficult to implement than blocking I/O, it often provides better performance and has become prevalent in modern servers [26].

Table 2. Experiments on testing the `java.nio` client API in JPF.

# tests	Model coverage		JPF states		other JPF statistics		
	# states	# trans.	new	visited	ins. [1,000s]	mem. [MB]	time
100	5	13	1,070	5	11,667	616	0:15
200	6	15	2,142	3	23,316	1,173	0:31
300	6	19	6,256	24	68,312	2,913	1:51
400	6	20	16,302	61	177,605	4,802	6:03
500	6	20	26,054	121	283,699	6,127	10:20

Table 3. Experiments on testing the `java.nio` server API in JPF.

# tests	Model coverage		JPF states		other JPF statistics		
	# states	# trans.	new	visited	ins. [1,000s]	mem. [MB]	time
100	7	17	676	0	7,185	294	0:16
200	7	17	4,798	223	52,128	551	1:10
400	7	17	6,917	230	74,823	1,131	1:46
800	7	17	11,973	273	128,910	2,607	3:37
1,200	7	17	14,760	282	158,872	4,396	5:33
1,600	7	17	20,194	338	217,039	5,996	9:57

We applied Modbat to three related models: one for the server API that focuses on accepting connections on an open port, and two for the client API that model connection usage and selectors, respectively. Out of these models, the server model is the smallest (having fewer states and less inherent non-determinism than the client models), and the client model that includes selector usage is the largest.

Modbat is first applied to our test model on the normal Java VM, taking the standard implementation of `java.nio` as a reference implementation. The model is written such that no (false) positives are reported in that case. Model execution in this case is very fast; for models without external communication, we measured 3,000 test per second on an 8-core Mac Pro workstation running Mac OS 10.7.

Execution in JPF is much slower than in the normal Java VM. The overall architecture remains the same, but Modbat and the SUT run inside JPF instead of the normal Java VM to analyze non-deterministic operations. While Modbat itself is deterministic, non-determinism in non-blocking I/O generates multiple possible successor states. When the outcome of a given test case is analyzed exhaustively (in JPF) rather than for only one execution (in the normal Java VM), coverage information maintained by Modbat diverges as well. This causes a state space explosion across multiple test executions (see Tables 2 and 3). In that table, the number of "visited" states in JPF indicates the effect of non-determinism on coverage information, and thus a super-linear increase in the state space.

Because of this growing memory usage by the state space generated JPF, generating a large number of test runs in a single execution is not possible. We

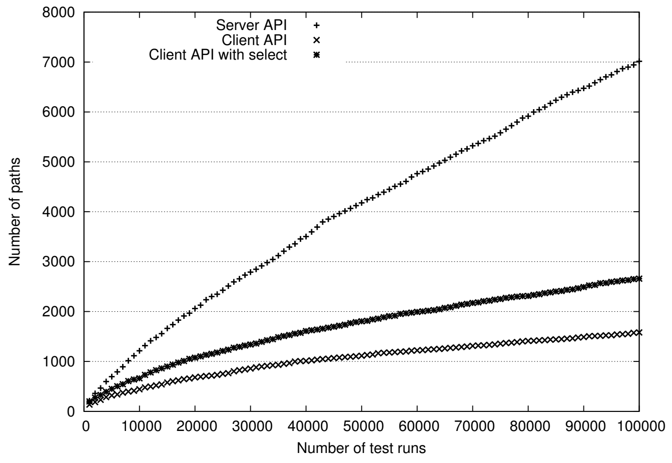

Fig. 9. Paths covered (out of infinitely many possible paths) during 100,000 test runs.

solve the problem by splitting 100,000 tests into small test suites, and multiple executions of Modbat in JPF. Taking advantage of our random number generator (see Section 3), we use the successor of the master random seed of the last test run in the next execution of Modbat. Coverage information is aggregated by post-processing the log files.

Full state and transition coverage was reached quickly in all models, after fewer than 1,000 test runs.[10] As the models include loops, the number of possible paths is infinite. However, most tests reach an end state after ten or fewer transitions. Due to this, the probabilistic test exploration of Modbat ends up covering previously derived test traces (see Figure 9). After 1,000 tests, between 13.7 and 21.3 % of all generated traces are unique; after 10,000 tests, that number drops to 4.4–12.2 %; after 100,000 tests, to 1.6–7.0 %. We consider these diminishing returns not to be a huge problem: Similar case studies have shown that a large number of significant SUT behaviors is covered after 2,000–5,000 cases [3].

5.4 Defects Found

We found two previously unknown defects in our `java.nio` library for JPF:

1. The wrong exception was thrown when `finishConnect` was invoked after `close`. This scenario is easily overlooked as manually written tests and (even our own earlier models) tend to focus on key operations of the SUT and neglect operations that come after "difficult" ones [25]. The bug was found

[10] We found that for a large number of models, state and transition coverage was even easier to obtain. However, a model state may reflect many system states, depending on the level of abstraction. Therefore, we think that more complex coverage metrics [1] are needed; their usage is future work.

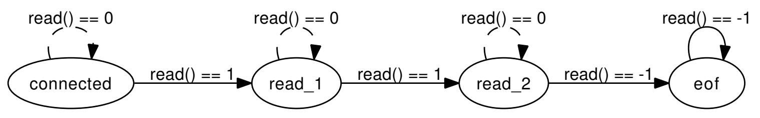

Fig. 10. Model of end-of-file semantics; dashed transitions are incomplete reads.

after introducing a short-hand in Modbat to model similar transitions more succinctly, made it expedient to include the scenario that triggered the defect. Manually written unit tests with exception checking code (see Figure 6) did not implement the exception check correctly, underscoring the importance of supporting such features with a DSL. From this we conclude that it is conceivable that a developer using other test tools (without DSL support) may have missed this bug.

2. It was possible to read spurious data after an end-of-file (EOF) token had been received. The test case for this includes a server that sends two bytes and then closes its connection. The model has to verify correct reception of data in the presence of possibly incomplete reads. The defect in the library related to a mismatch between the EOF event and its internal state when non-blocking reads are used. The problem was found thanks to improved monitoring code in the model. In this case, the property to be monitored was expressed programmatically, and was initially too weak. In hindsight, using extra model states and Modbat's `nextIf` feature would have expressed the same property more succinctly and clearly (see Figure 10).

From these case studies, we conclude that Modbat is effective at modeling APIs of I/O-driven systems. The fact that it allows to express certain features directly was related to finding defects that were not detected by manually written tests (because of faulty checking code) and earlier models (because of a less expressive DSL available then). Still, we also concur that creating a succinct and correct model of system states and actions as a compact EFSM is a challenge. The choice of appropriate model states and variables at the right level of abstraction requires experience and precise reasoning about the system specification.

6 Conclusions and Future Work

Modbat is a model-based tester based on extended finite-state machines. It differs from existing test generation tools by providing a domain-specific language that assigns system actions to transitions between model states. Modbat directly supports exception handling and non-deterministic system actions such as non-blocking input/output.

Our experience with Modbat shows that its notation is very versatile and allows the developer to focus on the system semantics and properties. Using Modbat, we successfully found previously unknown defects in a complex system. This was made possible because Modbat allows one to easily express a variety of system actions, including non-deterministic operations. The fact that previous attempts missed these defects suggests that model expressiveness is an important factor for the effectiveness of model-based testing.

Our current approach to test case derivation uses only a random seed as persistent state. This allows for parallelization of the search, but limits the number of unique paths found for a given number of test cases. In future work, we would like to investigate coverage-driven approaches, in a way that still allows splitting large test suites into small parts. We also plan to integrate Modbat with other test data generation tools, and consider searching finite models exhaustively.

Acknowledgements. This work is supported by *kaken-hi* grant 23240003. We would like to thank Dimitra Giannakopoulou, Falk Howar, Richard Potter, Rudolf Ramler, and Franz Weitl for their suggestions and feedback during our discussions.

References

1. P. Ammann and J. Offutt. *Introduction to Software Testing.* Cambridge University Press, New York, USA, 1 edition, 2008.
2. 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)*, 5717:279–286, 2009.
3. C. Artho, A. Biere, and M. Seidl. Model-based testing for verification backends. In *Proc. 7th Int. Conf. on Tests & Proofs (TAP 2013)*, LNCS, pages 39–55. Springer, 2013.
4. 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, 2013. IEEE Computer Society. To be published.
5. C. Artho, W. Leungwattanakit, M. Hagiya, and Y. Tanabe. Efficient model checking of networked applications. In *Proc. TOOLS EUROPE 2008*, volume 19 of *LNBIP*, pages 22–40, Zurich, Switzerland, 2008. Springer.
6. K. Beck. *Extreme programming explained: embrace change.* Addison-Wesley Longman Publishing Co., Inc., 2000.
7. K. Beck. Test driven development: By example, 2002.
8. K. Cheng and A. Krishnakumar. Automatic functional test generation using the extended finite state machine model. In *Proc. 30th Int. Design Automation Conference*, DAC 1993, pages 86–91, New York, NY, USA, 1993. ACM.
9. K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *SIGPLAN Not.*, 35(9):268–279, 2000.
10. J. Forrester and B. Miller. An empirical study of the robustness of Windows NT applications using random testing. In *4th USENIX Windows System Symposium*, pages 59–68, Seattle, USA, 2000.
11. G. Fraser and A. Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276 –291, feb. 2013.
12. E. Gansner and S. North. An open graph visualization system and its applications to software engineering. *Software—Practice and Experience*, 30(11):1203–1233, 2000.

13. P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, June 2005.
14. G. Holzmann. *The SPIN Model Checker.* Addison-Wesley, 2004.
15. J. Jacky, M. Veanes, C. Campbell, and W. Schulte. *Model-Based Software Testing and Analysis with C#.* Cambridge University Press, 1 edition, 2007.
16. 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)*, volume 80 of *EPTCS*, pages 58–72, 2012.
17. T. Kitamura, T. Do, H. Ohsaki, L. Fang, and S. Yatabe. Test-case design by feature trees. In *Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2012).*, volume 7609 of *LNCS*, pages 458–473. Springer, 2012.
18. J. Kramer. Is abstraction the key to computing? *Commun. ACM*, 50(4):36–42, April 2007.
19. J. Link and P. Fröhlich. *Unit Testing in Java: How Tests Drive the Code.* Morgan Kaufmann Publishers, Inc., 2003.
20. G. Myers. *Art of Software Testing.* John Wiley & Sons, Inc., 1979.
21. R. Nils. ScalaCheck, a powerful tool for automatic unit testing, 2013. https://github.com/rickynils/scalacheck.
22. M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: A Comprehensive Step-by-step Guide.* Artima Inc., USA, 2nd edition, 2010.
23. Oracle. *Java Platform Standard Edition 7 API Specification*, 2013. http://docs.oracle.com/javase/7/docs/api/.
24. C. Pacheco and M. Ernst. Randoop: feedback-directed random testing for Java. In *OOPSLA 2007 Companion*, Montreal, Canada, 2007. ACM.
25. R. Ramler, D. Winkler, and M. Schmidt. Random test case generation and manual unit testing: Substitute or complement in retrofitting tests for legacy code? In *36th Conf. on Software Engineering and Advanced Applications*, pages 286–293. IEEE Computer Society, 2012.
26. W. Reese. Nginx: the high-performance web server and reverse proxy. *Linux Journal*, 173, 2008.
27. K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. *SIGSOFT Softw. Eng. Notes*, 30(5):263–272, September 2005.
28. M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach.* Morgan Kaufmann Publishers, Inc., San Francisco, USA, 2006.
29. 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*, volume 4949 of *LNCS*, pages 39–76. Springer, 2008.
30. W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, 2003.
31. D. Wampler and A. Payne. *Programming Scala.* O'Reilly Series. O'Reilly Media, 2009.
32. A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *Software Engineering*, 28(2):183–200, 2002.