

With an Open Mind: How to Write Good Models

Cyrille Artho¹, Koji Hayamizu¹, Rudolf Ramler², and Yoriyuki Yamagata¹

¹ RISEC, AIST, Amagasaki, Japan

² Software Competence Center Hagenberg, Hagenberg, Austria

Abstract. Writing effective models for systems and their environment is a challenge. The task involves both mastering the modeling tool and its notation, and faithfully translating all requirements and specifications into a complete model. The former ability can be learned, while the latter one is a continuous challenge requiring experience and tools supporting the visualization and understanding of models. This paper describes our experience with incomplete models, the types of changes that were made later, and the defects that were found with the improved models.

Keywords: Model-based analysis, model design, model checking, model-based testing

1 Introduction

Model-based techniques use abstract models to define many possible system behaviors. In *model-based testing*, a test model gives rise to many concrete test cases. In *model checking*, all possible behaviors of a given model are explored to check if given properties hold. Both types of analysis have in common that a model of the environment is needed, which represents possible stimuli to the system under test (SUT). Analysis of the SUT involves exploring interactions between the SUT (model) and the environment, and verifying if a set of stated properties holds for all possible interactions (see Figure 1).

When using testing (run-time verification), tests can be executed against the implementation of the system. In model checking, a model of the SUT is needed; that model may be written by an engineer, or a tool may derive the system model from its implementation. In either case, the environment needs to be modeled from requirements, which is a largely manual task.

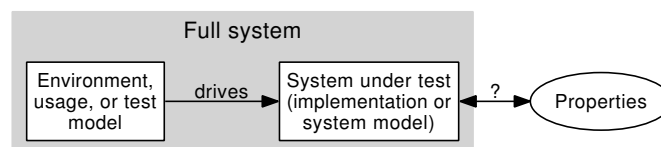


Fig. 1. Verification of a system in its environment.

The resulting model should reflect the requirements and capture all relevant system behaviors. Creation of a good model is a challenge, both for modeling the system and maybe even more so for modeling its environment.

If a property is violated by a given execution trace, then the trace is analyzed to determine whether the model is incorrect or the SUT contains a defect. As long as such counterexample traces are found, there is an inconsistency between the stated properties and the possible state space, as determined by the model. Defect resolution may involve relaxing a property that is too strict, restricting an environment model that is too general, or fixing a defect in the SUT. No matter which artifact in the verification process is at fault, property violations always show that development is not complete yet.

Unfortunately, it is a common fallacy to believe that if a model is analyzed exhaustively, a system has been “proven correct” if no defects are found by the analysis. There are many subtle ways in which a model may be too restrictive, or a stated property too weak, to cover possible defects. This paper describes our experience with this problem, and suggests steps to be taken to improve the situation.

To highlight this issue, we label the right arrow in Figure 1 with a question mark. We think that the common notion of “system *models* properties” as a verification goal is a good one. However, the notation is often thought of in the reverse direction as “properties hold for the system” once verification is complete. The danger in that notation lies in the fact that without *validation* of the model, property verification guarantees little in terms of system correctness.

Good models are those that help to exhibit defects in the system under test. What are the problems that restrict a model’s defect detection capability?

In this paper, we first describe various projects in which initial models were insufficient to find certain defects. However, even small changes or additions to these models uncovered many additional features. We identify factors that negatively influenced the design of the original models, and propose both procedural as well as technical remedies.

This paper is organized as follows: Section 2 describes related work. Our experience with software test models is described in Sections 3 and 4, while Section 5 shows a discrepancy between models and reality in hardware. Section 6 discusses our findings, and Section 7 concludes and outlines future work.

2 Related Work

In hardware and software analysis, properties may be trivially fulfilled, because not all relevant parts of a system have been analyzed, due to an incomplete system or environment model.

In hardware analysis, the problem of properties being trivially true has been well-known for two decades [4,5]. So-called *vacuous* properties include implications of type $a \rightarrow b$, where the antecedent a is never true. Regardless of the value of b , such a property holds. However, because the second part of the for-

mula becomes irrelevant, this case of an “antecedent failure” is likely not what the modeler intended [4].

For temporal logics, so-called *zeno-timelocks* describe cases where parts of a model would have to execute infinitely fast (being subject to an infinite number of transitions in a finite amount of time) for a property to hold [8]. Such timelocks often relate to a problem in the way parts of a system are modeled [9].

More recently, a different case, parts of a property that are unsatisfiable per se, has been investigated [20]. This property can be used to “debug” a specification, i. e., to analyze why a specification does not hold. There is emerging work in the field of diagnosing model checker specifications using scenarios to analyze the model [17].

In software testing, modified condition/decision coverage (MC/DC) and similar test coverage criteria try to ensure that each part of a complex conditional statement is actually relevant for the outcome of a test suite [1,26]. For each location in the software code where compound conditionals exist, MC/DC demands that, among other criteria, each condition in a decision is shown to independently affect the outcome of the decision [26]. If a condition has no effect on the outcome of a decision, it is likely incorrect (too weak) or redundant. The application of coverage criteria on the model level is emerging work, with only a few relatively simple coverage criteria such as state, transition, and path coverage, being commonly used so far [1].

Work investigating how human developers write test sequences has found that there is a bias towards designing a test case up to a “critical” operation that is expected to possibly fail, but not beyond [10,18]. In particular, test cases are often designed to cover possible exceptions, but tend to stop at the first exception. This bias was confirmed in our case studies for designing models for network libraries [2,3] and is described in more depth below.

Finally, the problem of model validation is also well known in model-driven engineering [6]. In that case, the model cannot be verified but only validated; recent work suggests generating questions about model properties as a form of sanity check [6].

3 Modeling the Java Network Library with Modbat

Even in widely used commercial software such as the Java platform [15], the official specification and documentation is written in English and not available as a fully rigorous formal document. This may give rise to ambiguities. In our experience, the biggest challenge in using the given specification was that many details are implicit, making it difficult to create a faithful model that also covers all relevant behaviors.

3.1 Setting

This section concerns the use of Modbat, a model-based test tool [2], for verifying a custom implementation of the `java.nio` network application programming interface (API) [15]. This network library allows the use of non-blocking

input/output (I/O) operations. Unlike blocking operations, which suspend the current thread until the full result is obtained, non-blocking variants return a result immediately; however, the result may be incomplete, requiring that the operation be retried for completion.

The goal of this project was to test conformance of a custom version of the `java.nio` library [3] w.r.t. the official documentation [15]. The custom implementation of the `java.nio` library is designed to run on Java PathFinder [25], which requires a model implementation of code that interacts with the environment [3]. When using Modbat on this library, the model replaces the environment and generates calls to the API of the SUT.

Modbat uses an extended finite state machine [23] as its underlying model. State transitions are labeled with *actions* that are defined as program code (functions implemented in Scala [14]). This program code can directly execute the system under test (in our case, parts of the Java API). In addition to that, Modbat also supports exception handling, by allowing a declaration of possible exceptions that may result by (failed) actions. Finally, Modbat supports non-blocking I/O by allowing the specification of *alternative target states* to cover both the successful and the failed (incomplete) outcome of non-blocking I/O.

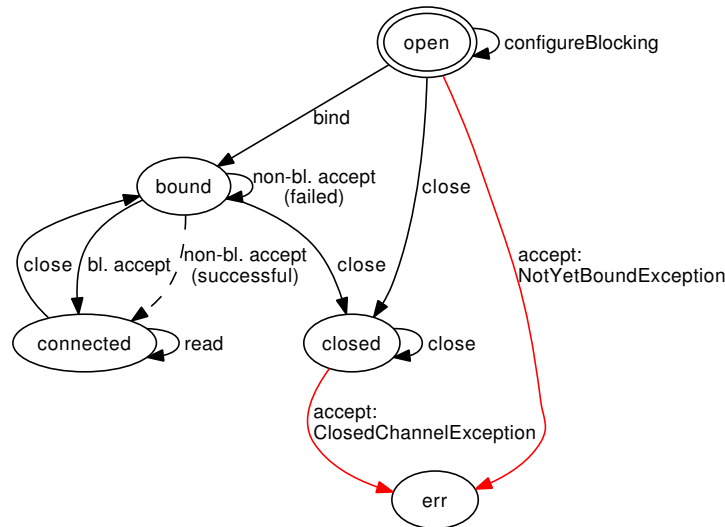


Fig. 2. Initial model for `java.nio` server API.

We have modeled the usage of the key classes `ServerSocketChannel` and `SocketChannel` with Modbat (see Figure 2 for the server case). Both APIs have in common that a channel object first needs to be created by calling `open`. Our models take the resulting state as the initial state. In the server case, the created object represents the ability to accept incoming connections; the object

therefore also needs to be bound to a port and IP address before a connection can be accepted. In the client case, the connection can be established directly by supplying the IP address and port of the server as a function argument. However, the client API is slightly more complex in general in the sense that finishing a pending connection (after an unsuccessful non-blocking `connect` call) attempt requires a different function than the initial attempt, viz., `finishConnect`. There are also more possible exceptions [15].

In the figure, dashed transitions correspond to the successful (completed) case of a non-blocking operation that would otherwise have to be repeated (non-blocking `accept`). Red, accordingly labeled edges correspond to exceptions resulting from actions that are not allowed in a given state. In these cases, the edge label denotes the exception type. Some nodes have a self-transition that denotes a possible switch from blocking to non-blocking mode using `configureBlocking`. A self-loop may also denote a retry of a previously failed non-blocking action; in the successful case, the dashed alternative transition is taken to the *connected* state. Finally, there is a self-transition in the connected state that reads from the newly connected channel before the connection is closed again.

3.2 Weaknesses of the initial model

We first executed the test cases generated from the models against the standard Java implementation, using it as a reference implementation. This ensures that no false positives are reported by the test model when it is used as an oracle against the reference implementation. We then used the given test model in a second test run, against our network model for JPF. Using this approach, we found a complex defect that was not covered with manually written tests [3]. However, several defects were not discovered by this initial model.

First, the initial model did not cover all possibilities of disallowed operations in the closed state. In that state, only `close` is allowed, as its semantics is defined to be idempotent in Java [15]. All other operations are expected to throw a `ClosedChannelException`. This part of the semantics is trivial to model, because most operations behave identically. However, the initial model missed several possible alternatives, because they have to be enumerated by the modeler. As it turned out, the implementation did not track its internal state correctly in all cases, and the wrong type of exception was thrown for a particular sequence of commands that included `close` and another operation after `close`. The challenge is that the model has to cover a large number of possible transitions, and it is easy to overlook some.

Second, it was difficult to express a property related to an end-of-file return code correctly [2]. An older version of the model was using a precondition to avoid reading from a stream where an end-of-file has been received. This meant that sequences that attempt to read beyond the end of a file were never generated, missing a defect in that case. A newer model included such sequences but its property to be checked was a bit too lenient. The reason for this was that it is not trivial to account for all possibilities of reading data in non-blocking mode. Even for an input of very limited length (2), the state space of all possibly incomplete

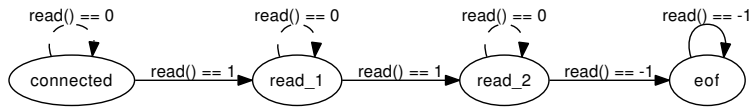


Fig. 3. Model of end-of-file semantics; dashed transitions are incomplete reads.

read operations that eventually lead to the end-of-file is quite large (see Figure 3). The property was initially written programmatically, and the code did not track the internal state strictly enough under all possible circumstances; Figure 4 shows how a unit test that includes repeated calls to `readByte` and checks the result in a case where the input has length 2. Most of the code, including a counter and a flag, is devoted to expressing the property. As Figure 3 shows, a finite-state machine can express the property much more succinctly [2].

```

SocketChannel sc = connect();
sc.configureBlocking(false);
int n = 0;
boolean eof = false;
while (n < 4) {
    int read = readByte(sc);
    if (eof) assert (read == -1); // ensure no 0 non-bl. read after EOF
    if (read == -1) eof = true;
    if (read != 0) { // ignore non-bl. zero-reads
        if (n++ < 2) { // read data twice
            assert (read == 1);
        } else { // always read EOF after that
            assert (read == -1);
        }
    }
}
sc.close();

```

Fig. 4. Unit test including the end-of-file property.

Third, the initial model was also limited in that it included an error state for all cases where an exception has occurred [2]. This limits test cases to execute only up to a possible exception, but not beyond it. The reasoning behind this is that a well-behaved user of a library *never* triggers an exception due to incorrect use, of the types specified in the model. However, a component (an object provided by the SUT) can usually survive an incorrect command by refusing to execute it and throwing an exception instead. Because of this, it is possible to continue a test beyond such a step, issuing more correct or incorrect commands. This situation tends to be overlooked when modeling the environment of a system. Earlier case studies have shown that this is a common human bias in testing [10,18], and this has also carried over to modeling. While there indeed exist common cases where an object cannot be used after an exception has

been thrown, this is not the case for incorrect operations used on communication channels.³ In the updated server model (see Figure 5), a trace can include other operations (or the same one) after a given operation resulted in an exception.

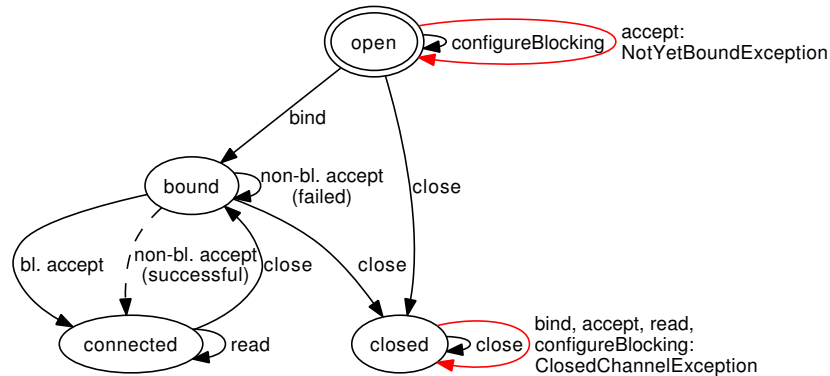


Fig. 5. Improved model for java.nio server API.

3.3 Summary

We found three problems with an initial model for a complex API in Java. The first problem was caused by the model not including all possible alternatives. The second problem was caused by a property that cannot be easily expressed in code, but where a finite-state machine may capture its semantics succinctly. Finally, the third problem stemmed from a human bias that tends to focus on operations up to a “critical” operation throwing an exception, which lead to the model being restricted by using an error state as a global target state for all exceptions. Instead, a self-loop should have been used to allow traces to continue beyond an exception.

4 Experiences with Models for Testing Collection Classes

4.1 Setting

In a series of experiments we studied the effectiveness of tool-supported test case generation in comparison to humans developing unit tests [18,19]. The experiment was based on a library of collection classes (i.e., containers such as list, array, set, stack, and map) with manually seeded defects. The library we used resembles the common Java collection classes. Thus, the study material had

³ On the other hand, a communication channel is usually in an unrecoverable state after an exception thrown due to an I/O failure.

the benefit of being well known by the study participants and did not require additional background information or familiarization. The size of the library was about 2,800 lines of Java code distributed to 34 classes and interfaces and a total of 164 methods. Most classes showed high algorithmic complexity and used object-oriented concepts such as interfaces, abstract classes, inheritance, polymorphism and dynamic binding.

In this context, we also briefly looked into the possibilities of model-based approaches for unit testing [23] and developed some preliminary ideas about how to construct models for our collection classes. However, model-based testing was not part of one of our studies so far and, thus, the initial models have not been evaluated further. In the following, we document our observations about some of the challenges involved in constructing these initial models.

4.2 Modeling Test Scenarios

The starting point of our modeling attempts was the focus on developing unit test suites for the collection classes. The main motivation was to reduce the manual effort involved in implementing unit tests by automatically generating some or all of the test cases. So our initial perspective on modeling was influenced by the ideas and scenarios we wanted to explore in unit testing.

One of the first models was, thus, a generalization of a specific scenario that can be implemented as simple unit test. The objective of this test was to add and remove elements to/from a collection and to check the corresponding size of the collection as well as that the removed elements were those that had previously been added (Fig. 6).

```
@Test public void testAddRemove() {
    LinkedList l = new LinkedList(); assertEquals(0, l.size());
    assertTrue(l.add("1"));           assertEquals(1, l.size());
    assertTrue(l.add("2"));           assertEquals(2, l.size());
    assertTrue(l.remove("1"));        assertEquals(1, l.size());
    assertTrue(l.remove("2"));        assertEquals(0, l.size());
}
```

Fig. 6. Exemplary unit test capturing a specific sequence of add/remove operations.

This test implements one specific, representative case out of the many possible sequences in which elements may be added and removed. The model we initially developed still focused on the particular scenario of adding and removing elements (see Fig. 7).

Yet with the help of this model, we were able to generate a huge set of test cases that covered a wide range of combinations in which elements were added and removed, eventually including also all the combinations implemented in the manually developed test cases. Several other scenarios (e. g., using iterators or sorting) were modeled in the same way, again with the intention to explore them more extensively with huge sets of generated tests.

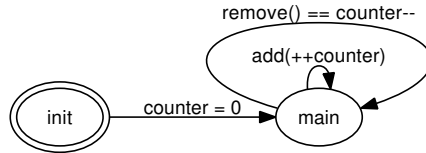


Fig. 7. Simple model for generating arbitrary sequences of add/remove operations.

A weakness all these initial models had in common was that no further defects were revealed, other than those that already had been found by the manually written unit tests. By simply transferring the unit test scenarios into test models, the resulting models were inherently limited to the underlying scenarios. Since in our case these scenarios were already sufficiently covered by a (small) set of manually implemented unit test cases, the model-based testing approach was not able to reveal new errors.

4.3 Modeling Actions on the System Under Test

To improve the initially generated test cases and to better unleash the potential of the model-based testing approach, we tried to advance the models towards more realistic usage patterns. For example, we added further actions to include all methods exposed in the public interface of a collection class and we removed the guards preventing invalid method calls to cover a broad range of interactions in addition and in combination to adding and removing elements. We also integrated the different small models into one large model. For example, we integrated the model testing iterators to make sure several iterators were used as part of a larger scenario where collections are modified concurrently. The advanced models actually generated new fault-revealing test cases we were not thinking about when manually writing tests. Eventually, thus, we reached the conclusion that the most realistic results will be achieved by developing a model that resembles the complete system under test as closely as possible in favor of developing several more specific models that reflect only individual test scenarios.

So far we have not completed the development of a full model for the collection classes. Nevertheless, we found that partial models or models at a higher level of abstraction are already capable of detecting some of the faults, although they are not rigorous enough to detect all the faults. Incrementally developing and refining the models provides the benefit of early defect detection and allows to balance the invested modeling effort to the achieved test results.

When proceeding towards a more complete model, we encountered another challenge that still remains an open issue. With the exponentially increasing number of possible scenarios described by a large model, the probability to sufficiently cover all the known interesting scenarios and critical corner cases tends to decrease. For example, since `add` and `remove` are equally probable, the number of elements in a collection usually stays low and collections rarely grow to

the point where new memory is allocated. Another example is related to the special cases when inserting and removing list elements; the first and the last position of a filled list have to be treated differently and should to be covered by dedicated test cases. Yet this knowledge is not part of the model. However, since this knowledge is already available when creating the model, it would be useful to include it at this point as an aid to guide test case generation in direction of the relevant scenarios.

4.4 Modeling Test Data

Models of collection classes usually exhibit only a small number of relevant states. We found that an important aspect of the model relates to the test data, i. e., the data elements to be stored in the collections.

Our initial model concerning the add and remove operations used a counter *nrOfElements* to keep track of the size of the collection and to compute its state, i. e., empty or filled. When adding an element to the collection, we used the counter as new integer object to be added. When removing an element, we compared the obtained element with the counter to make sure the expected element had been returned. Thus, this simple mechanism dynamically generated reproducible test data. To avoid that the sequential order of the elements derived from the counter created unbalanced sequences, e. g., new elements are always added to the end of the collection, we used the counter as seed for a random number generator.

A weakness of this first model was that it missed errors caused by mixing data elements of different type. The containers *TreeSet* and *TreeMap* are sensitive to such errors as are the operations for sorting and searching in collections. Thus, the initial model did not find the related seeded defects since only comparable data objects of type *Integer* were used.

We extended the initial model by creating numerous test data elements of different types when setting up the model. The data elements were stored in an array in arbitrary order. The counter we previously used in the model now served as array index, which still allowed to determine the expected element to be returned by a remove operation.

Only later we found a new fault that indicated that there is still room for further improvement. The implementation of the collection classes was not able to handle the case of a collection being added to itself. Some operations such as `toString` would then lead to an unbounded recursion (see Fig. 8). We further extended the model to dynamically add new data elements to the test data set not only at startup but also while the model is executed. In future we plan to extend the model to incorporate the idea of feedback-directed test generation [16].

```
Stack s = new Stack(); s.push(s); s.toString();
```

Fig. 8. Sequence revealing an unbound recursion in the implementation of `Stack`.

A related issue is involved in using `null` values, since the implementation of some container classes accept `null` as valid data elements whereas others do not. This issue was found when we tried to reuse generic models for different container classes. This observation led us to the (ongoing) discussion to what extent a model should reflect the behavior of the system under test versus its environment, i. e., the allowable inputs from the anticipated usage. While Utting et al. [24] classify this scope as a binary decision (input-only versus input-output models), we found that our models always combined both sides since modeling the input side also required some knowledge about the expected output.

4.5 Summary

We reported on work on modeling the behavior of Java container classes. Initial models that were created from generalizations of existing unit tests ended up not being effective at finding defects that were not already covered by unit tests. When extending these models, we found that models that are convenient to define (for example, using only numbers) end up not covering important cases such as different data types or `null` values. Finally, creating modular and reusable models is difficult, because small differences in components result in pervasive differences in the allowable inputs requiring extra effort to adjust.

5 Industrial Project: Electric Circuit

5.1 Adapted work flow

As described in Section 2, various subtleties regarding the aspects of timed models and the reachability of model and system states in hardware are known. To avoid incorrect models, verification engineers validate their model with domain experts, who design the circuit. In this project, we employed the following work flow to eliminate false positives (spurious warnings) and false negatives (missed defects):

1. For a set of given desired states, reachability of these states is checked. For example, any terminal state in the system should be reachable.
2. The specification is negated and model checked. This means that the model checker analyzes whether there exist paths in the model that fulfill the desired property. In a correct model, correct execution paths should be generated. These execution paths are generated as counterexamples by the model checker, as the real property has been negated. Different counterexample paths are subsequently reviewed together with domain experts to determine whether they are correct and reflect the expected behavior of the system.
3. Properties that are trivially expected to hold are checked as well, as a form of sanity check.

5.2 Problem found

The work flow described above prevents many defects in the model. However, despite this, a modeling problem was found in an industrial project on an electric circuit. The problem is related to how time is modeled in a real system. The system model uses discrete time, where the state of each component is updated on the next model clock tick. However, in real hardware, components can change their state almost immediately; the “slowness” introduced by discrete time gave rise to a counter-example in this model (see Figure 9). The problem in this model is that `gSet_PT_Voltage_A01` is updated in the next state even though the voltage change is immediate in real hardware.

```
next(gSet_PT_Voltage_A01) := case
  Gen_to_Load_A = 0 & Brk_A = 0 & GL_V_A = 0 & LG_V_A = 0 : V_Emp;
  Gen_to_Load_A = 0 & Brk_A = 0 & GL_V_A = 0 & LG_V_A = 1 : V_Lrd;
  Gen_to_Load_A = 0 & Brk_A = 0 & GL_V_A = 1 & LG_V_A = 0 : V_Gen;
  ...
esac;
```

Fig. 9. Part of a model transition describing an industrial circuit.

After the counterexample was investigated together with domain experts, it was considered to be spurious (a false positive). To fix the model, the first line in the model was amended to `gSet_PT_Voltage_A01 := case` (i. e., `next` was removed). This eliminated the false positive.

6 Discussion

We have reported our experience from several modeling projects. In each project, there were unexpected problems with creating a correct and sufficiently good model to fulfill the purpose of model-based verification. In our opinion, it is interesting that the problems were not caused by ambiguities of the requirements or documentation. Where ambiguities caused problems, we were able to identify them and clarify the open points by checking the reference implementation.

6.1 Model design

In our projects, problems arose when requirements were transformed into a model. We often failed to create a model that matches a wide range of all possible behaviors stated in the requirements. All of the models were “correct” but failed to cover certain behaviors of the system, some of which were even implemented incorrectly. In the software projects, the uncovered behaviors resulted in missed defects (false negatives); in the hardware project, it resulted in a spurious error (false positive), which gives an indication of a mismatch between the model and reality.

The lack of expressiveness in the models did not originate from unintended errors or oversights, but from intentional abstractions or decisions that led to elegant models. The resulting lack of coverage was therefore a side-effect of conscious design decisions. From this observation, we identify the right level of abstraction and human bias [10] as the key problems.

Abstraction. A major problem of creating a good model is to choose the right level of abstraction. This is a very difficult problem that takes years of experience to solve well. Some people even claim that this skill may not be teachable but an innate ability [12]. In the future, we expect (modeling/abstraction) teaching methods, and design tools, to improve to make the task a bit less daunting.

Human bias. When choosing an abstraction, human bias also often exists in that the model is designed for a narrower purpose than necessary. This leads to the omission of certain behaviors in the model. Like abstraction, this is a fundamentally difficult problem to overcome; it requires to attack the problem from various angles to obtain a comprehensive solution. We think that involving a team of people in modeling, and making an effort to avoid any preconception, can at least mitigate this problem. Ideally, models are created with an open, fresh mind, and no possibilities, regardless of being difficult or trivial to handle, should be disregarded. In practice, this may require careful engineering of the model w.r.t. code reuse, if one takes into account that many small subtle differences in system components result in a large increase of different possible behaviors (and thus models or parameterizations thereof).

6.2 Model validation

Our experience shows that model-based verification has to be grounded in an extensive validation of the model. Even though validation is in itself not a fully mechanized activity, there exists tool support for tasks such as coverage analysis and visualization, which contribute to validation. Furthermore, computer support can also be used in the modeling stage itself if certain artifacts such as a reference implementation are available.

Machine learning. Machine learning of models has the obvious advantage of not missing system states due to a simple oversight. If a correct (reference) implementation of a system exists, then a model can be derived from the existing system using machine learning [21,13]. The resulting model may not be human-readable but its verification verdict may confirm or refute the result obtained from a model designed by a human.

This approach can even be used if it is not known if a given system is correct; in that case, the model reflects its current (possibly not fully correct) behavior and can be used in regression testing to see if the behavior of the system changes in unexpected ways. Changes that violate a given property then would likely be found by a model that reflects the semantics of an older (“known good”) version of the system.

Diagnosis and visualization. Some of the weaknesses observed in models arose from the fact that they were generalizations of existing test scenarios. Therefore, just lifting a set of execution traces to a grammar-based model is not guaranteed to add much value. It is also necessary to check whether the existing test scenarios, and the derived model, are comprehensive enough.

We therefore advocate that model-based verification be combined with model diagnosis and visualization, so possible flaws in a model learned from an incomplete set of tests, or a defective system, may be found. In our hardware project, we already had adapted such a workflow by checking a sample of all possible execution traces generated by the model checker.

Model coverage and mutation analysis. It is important to analyze the *coverage* of the model in the real system; this can be done for software testing in a straightforward way [1,26]. However, coverage analysis on the final product gives us limited information on the expressiveness of the model itself (in addition to not being able to tell us whether all requirements are actually met). Hence, we also advocate mutation operators for models to find mutants that still pass the properties.

Mutating model properties is well-known [7] (and very similar to program code [11]). However, work needs to be done on mutating the structure of the model: a model could also be mutated by duplicating or deleting a transition, or changing its source or target state. This reflects what we have learned from our software model, where the model structure itself (and not just a given predicate or property) restricted its behavior.

Combination of model-based and model-free techniques. When analyzing the implementation of a system, fully automated analysis techniques can complement human efforts. Unlike in the case where a model is designed by a human, automated techniques have no test oracle that evaluates the output of the analyzed behaviors; instead, they serve as a “sanity check” for a wide range of generic properties (accessed memory must be initialized, no deadlocks, etc.).

When using such “model-free” techniques, randomized testing [16] often finds defects that humans miss [18]. Defects found by such tools may in turn spur an improvement in a manually written model. A comparison of the states covered by model-free techniques with the coverage of a manually written model, may unveil weaknesses in the latter as well.

7 Conclusions and Future Work

Writing good models is a challenge. Models should not only be correct but sufficiently expressive and inclusive to fulfill the purpose of finding defects or ensuring their absence. Finding the right level of abstraction, and trying to avoid human bias, or two of the key challenges in this process. A high level of abstraction that allows an efficient encoding of a model (or reuse of existing model code) may not cover enough details of all possible behaviors. Modeling languages and

tools should strive to improve this trade-off. Teaching engineers about commonly encountered problems or human bias is also essential.

We have listed several non-trivial flaws that we found in existing model development projects, and we have given suggestions how these may be avoided in the future. Tool-supported analysis of the model itself will help to explore the system behavior in its full breadth and may uncover missing model aspects that human inspection misses. In this context, we advocate using model-free, automated approaches where possible, so that their coverage can be compared with the coverage yielded by a model derived from the specification.

We believe that more case studies can also shed more light into why certain properties tend to be forgotten, and what types of modeling challenges engineers typically encounter. This may eventually lead to the creation of a body of knowledge for modeling and its effective use in practice. Existing work tends to focus on surveying approaches and tools, such as MCBOK, a body of knowledge on model checking for software development [22]; we hope that more fundamental cognitive and process-level issues will also be covered in the future.

Acknowledgments. We would like to thank Takashi Kitamura and Kenji Taguchi for their suggestions on this paper.

References

1. P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, USA, 1 edition, 2008.
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)*, LNCS, Haifa, Israel, 2013. Springer. To be published.
3. C. Artho, M. Hagiya, R. Potter, Y. Tanabe, F. Weigl, 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.
4. D. Beatty and R. Bryant. Formally verifying a microprocessor using a simulation methodology. In *Proc. 31st Conf. on Design Automation (DAC 1994)*, pages 596–602, San Diego, USA, 1994.
5. I. Beer, S. Ben-David, C. Eisner, and A. Landver. Rulebase: An industry-oriented formal verification tool. In *Proc. 33rd Conf. on Design Automation (DAC 1996)*, pages 655–660, Las Vegas, USA, 1996.
6. A. Bertolino, G. De Angelis, A. Di Sandro, and A. Sabetta. Is my model right? let me ask the expert. *J. Syst. Softw.*, 84(7):1089–1099, July 2011.
7. P. Black, V. Okun, and Y. Yesha. In E. Wong, editor, *Mutation testing for the new century*, chapter Mutation of model checker specifications for test generation and evaluation, pages 14–20. Kluwer Academic Publishers, Norwell, MA, USA, 2001.
8. H. Bowman. How to stop time stopping. *Form. Asp. Comput.*, 18(4):459–493, 2006.
9. H. Bowman, G. Faconti, J-P. Katoen, D. Latella, and M. Massink. Automatic verification of a lip synchronisation algorithm using UPPAAL. In *Proc. 3rd Int.*

- Workshop on Formal Methods for Industrial Critical Systems*, pages 97–124. CWI, 1998.
10. G. Calikli and A. Bener. Empirical analyses of the factors affecting confirmation bias and the effects of confirmation bias on software developer/tester performance. In *Proc. 6th Int. Conf. on Predictive Models in Software Engineering*, PROMISE 2010, pages 10:1–10:11, New York, NY, USA, 2010. ACM.
 11. Yue J. and Mark H. An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on*, 37(5):649–678, 2011.
 12. J. Kramer. Is abstraction the key to computing? *Commun. ACM*, 50(4):36–42, April 2007.
 13. A. Memon and B. Nguyen. Advances in automated model-based system testing of software applications with a GUI front-end. *Advances in Computers*, 80:121–162, 2010.
 14. M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Inc., USA, 2nd edition, 2010.
 15. Oracle. *Java Platform Standard Edition 7 API Specification*, 2013. <http://docs.oracle.com/javase/7/docs/api/>.
 16. C. Pacheco, S. Lahiri, M. Ernst, and T. Ball. Feedback-directed random test generation. In *Proc. 29th Int. Conf. on Softw. Eng.*, ICSE 2007, pages 75–84, Washington, DC, USA, 2007. IEEE Computer Society.
 17. I. Pill and T. Quaritsch. Behavioral diagnosis of LTL specifications at operator level. In *Proc. 23rd Int. Joint Conf. on Artificial Intelligence (IJCAI 2013)*, Beijing, China, 2013. IJCAI/AAAI.
 18. 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.
 19. R. Ramler, K. Wolfmaier, and T. Kopetzky. A replicated study on random test case generation and manual unit testing: How many bugs do professional developers find? In *Proc. 37th Annual Int. Comp. Software & Applications Conf.*, COMPSAC 2013, pages 484–491, Washington, DC, USA, 2013. IEEE Computer Society.
 20. V. Schuppan. Towards a notion of unsatisfiable and unrealizable cores for LTL. *Sci. Comput. Program.*, 77(7-8):908–939, 2012.
 21. B. Steffen, F. Howar, and M. Isberner. Active automata learning: From DFAs to interface programs and beyond. *Journal of Machine Learning Research—Proceedings Track*, 21:195–209, 2012.
 22. K. Taguchi, H. Nishihara, T. Aoki, F. Kumeno, K. Hayamizu, and K. Shinozaki. Building a body of knowledge on model checking for software development. In *Proc. 37th Annual International Computer Software & Applications Conf. (COMPSAC 2013)*, Kyoto, Japan, 2013. IEEE.
 23. M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers, Inc., San Francisco, USA, 2006.
 24. M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing approaches. *Softw. Test. Verif. Reliab.*, 22(5):297–312, August 2012.
 25. W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, 2003.
 26. Y. Yu and M. Lau. A comparison of MC/DC, MUMCUT and several other coverage criteria for logical decisions. *J. Syst. Softw.*, 79(5):577–590, May 2006.