Separating Testing Concerns by Means of Models

Dirk Wischermann
Friedrich-Alexander University
Erlangen-Nuremberg
Systems Software Group
Erlangen, Germany
dw@cs.fau.de

Wolfgang Schröder-Preikschat
Friedrich-Alexander University
Erlangen-Nuremberg
Systems Software Group
Erlangen, Germany
wosch@cs.fau.de

ABSTRACT

The number of potential execution paths through software usually increases dramatically with the size of the program. However, many coding errors only appear while executing few *particular* paths. Thus, it is a challenge for software testers to select a feasible subset of all paths to be covered in order to find most errors. This article describes a way of using behavioural models (such as state diagrams) for separating concerns in structural testing. Each model describes one concern, such as a usage protocol, a policy or a more complex behaviour. The goal is to get a better and differentiated reliability testimony out of fewer test cases, to find bugs that would probably not manifest themselves otherwise and to provide helpful information for debugging. Opposed to many other approaches that target on a high automation level or on achieving synergies between design and test process, our approach allows for detecting more errors with the same test cases (by means of generated built-in tests) and for selecting better test cases (using *adequate* coverage criteria). Having to supply the required knowledge in form of models means shifting effort from testing to development.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Programming by contract*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing Tools, Tracing*

General Terms

Reliability

1. INTRODUCTION

Testing is done to provide confidence in the correctness of software. The two major sectors of testing, functional and structural testing, pursue different strategies: functional testing aims to show the fulfillment of requirements by covering a specification with tests, whereas structural testing gives evidence for the absence of errors by covering the code. In general, neither can be accomplished *completely*; that is why models are built. *Model-Based Testing* is applied in both fields: Functional testing is done with *Usage Models* which describe, how the software is expected to be used (e.g., with *Markov Chains*) or with *Test-Models* which describe input data and activities of the testing environment. Structural testing is done with *Software Models*[1] that describe the software itself. Our article is about structural component testing with behavioural software models.

1.1 Problem Statement

Structural testing with strong coverage criteria is in practice limited to the level of *Unit Testing*, where units refer to methods or functions. This is fine as long as each unit has a single, well defined responsibility. We know that this is not the case in general. A simple example of a concern that is often not covered by unit testing, is ensuring that a reusable resource (such as memory, a file or a lock) is always freed after using it.

Especially object-oriented code is, for the sake of variability, portability and reuse, often organised by tricky architectural and design patterns, which is difficult to overview even for experienced software engineers. Furthermore, the code is superimposed by crosscutting concerns that do not fit into the chosen modularisation.[2] This results in a high number of potential execution paths through the software. However, the detection of distributed errors, to which different code snippets contribute (such as temporal protocol violations or deadlock conditions), often depends on executing a *particular* path and only succeeds if this path is included in the chosen coverage criterion. This problem can be tackled by finding a feasible *and* adequate coverage criterion.

We presume that purely syntactically defined criteria are either not feasible or not strong enough for testing more complex code: then it becomes likely that test cases fulfil the coverage criterion without revealing particular errors. On the other hand, we know that not every line of code and every case distinction is relevant for meeting every functional requirement. This is where models come into play; they can help to select a relevant set of test cases with a high error detection rate.

[1]Some authors call them *System Models*, but we need to reserve the term *System* for deployed systems including the hardware.
[2]Techniques like AOP[16], FOP[4] or AML[1] aim to solve this problem for the programmers, but rather complicate testing (see [27, 23, 20]).

1.2 Basic Terms

Structural code coverage and model-based testing are both quite elaborated topics already and some terms are well-established. We explain the most important ones here; pointers to respective literature are given in Section 5. The most common code coverage criterion is *statement coverage* (also known as the C_0 criterion), that is that each statement must be executed at least once while testing. *Branch coverage* (C_1) requires test cases that pass through each branch of the code. The difference to statement coverage is that this criterion requires test cases that pass through **empty** `if` or `else` branches (or similar) of the code. Higher criteria are based on data flow (definitions and uses of variables) or require exhaustive testing of case distinctions. The strongest criterion is *path coverage* (C_∞) and requires the test cases to execute every path through the code. This criterion subsumes all other criteria [28] but has no practical relevance, since it requires to cover an infinite number of paths if the code contains loops. In this article, we allude to *k-path coverage* ($C_{\infty,k}$), which results from path coverage by limiting the number of required loop executions to k.

Regarding model-based testing, the terms are not so settled. Opposed to structural (*white box*) testing approaches, most model-based techniques are *black box* approaches that do not comprise knowledge about the implementation. *Grey box* testing, for instance proposed in [22], mostly means to measure the coverage of *model* entities, where the subject of the model is the code. Furthermore, we can differentiate *graphs* and *models*: Models imply some *abstraction* by either by encapsulating or by leaving out information. Graphs (such as control flow graphs) are predominantly a graphical representation of the software or model without introducing (for testing purpose relevant) abstraction.

1.3 Our Approach: Perspective Testing

With Perspective Testing we introduce a generic way of specifying additional information about code coherence at the level of single instructions by means of behavioural models. Such a model declares which slice of the source code contributes to one specific concern, even if its implementation is scattered over several methods and classes. We exploit the models in a twofold manner:

- Deriving *differentiated coverage* information from test case execution. We measure the code coverage with regard to relevance for every single concern. For example, this allows for demanding markedly high coverage for important concerns and therewith for controlling the test process in a well directed manner.
- Checking constraints on interrelated operations. We interpret the models as *behavioural contracts* and generate built-in test code (BIT) for surveying whether the contract is respected. As an example, calling methods on an object is only allowed in a certain order or while holding a lock.

Our approach resembles a human reviewer that analyses one concern by one, looking for relevant operations and checking their proper use. By using a number of different models, we can *factorise* the testing process along concerns instead of separating it along units. At this point, the testers have the advantage over the developers that have to assign a place to each line of code while replication is not wanted.

This factorisation partially inverts the combinatorial explosion that arises from the shared implementation of the concerns within a component. The effort for achieving conventional k-path coverage increases exponentially with the number of case distinctions; thus, the number of test cases is multiplied for each additional concern in the component under test. For achieving *individual* path coverage by factorisation, the effort is only additive.

The built-in test code, which we derive from the same models, complements that control-flow oriented testing by recognising erroneous usage or behaviour patterns. Such errors often leave faulty states behind that may become manifest only in a completely different execution context or much later. With BITs, we get closer to the actual defect.

Our contribution is a way to allow the software engineer (tester, programmer or system integrator) to specify these contracts and therewith the slice of code that is to be individually covered. The Perspective Testing approach is opposed to many model-based testing approaches (and tools) that support *structural model coverage* as test selection criterion – this means covering edges and nodes of the model [26]. This allows for drastically reducing the number of test cases to be executed, too, but we think that for gaining confidence in the correctness of the code, such coverage is of limited use. We contrast it with our idea to use the models for defining coverage criteria *on the code*.

The remainder of this article is organised as follows. We introduce the Perspective Testing models in Section 2 and explain them by means of a simple example. An overall impression of the interplay of different concerns is given by our demonstrator example, which is a parallel embedded webserver (Section 2.1). Sections 3 and 4 are about shaping adequate coverage criteria and generating BITs out of the models.

In the related work section, we oppose our idea to different comparable – static as well as dynamic – approaches, especially in the domain of system software. Since this is work in progress and our tool is under construction, we have no results yet. Instead, we discuss some practicability issues and an give an evaluation plan, before discussing some ideas and concluding.

2. MODELS AT CODE-LEVEL

Our Perspective Testing approach uses software models that describe the software itself. This is opposed to the use of testing models that rather describe the environment and are not suited for structural testing. We propose to use state machines as described below using UML-like syntax. We differentiate three different roles of models in our approach:

- A model $\mathcal{M}_A$ that defines the allowed behaviour.
- An design model $\mathcal{M}_D$ that concretises $\mathcal{M}_A$ and reflects the *intended* behaviour of the software.
- An implementation model $\mathcal{M}_R$ that reflects the actual realised behaviour.

$\mathcal{M}_A$ defines a valid behaviour that must not be violated by the implementation. It consists of states and transitions with *operations* as trigger conditions, enriched by information about operations that are *relevant* for the modeled concern. It includes for each state the information, which operations are allowed or not. This information can be given

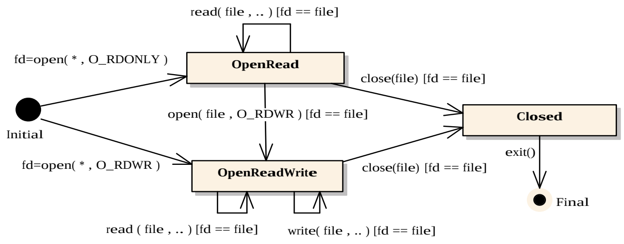

Figure 1: Model for the concern `HandleFile`. The file-descriptor serves as identifier and as global guard.

implicitly by defining that the *mentioned* operations are relevant (see Figure 1) or by explicit lists as in Figure 2. In addition to encoding information in states, data can be stored in variables and be accessed by guard expressions.

We say that an implementation *violates* a model iff a relevant operation is executed where it is not allowed. $\mathcal{M}_A$ can be reused for testing the same concern in several components, for example by a systems integrator for revealing API-misuse. In our example, calling `exit()` while a file is open would violate that model.

If only a part of the allowed behaviour is needed or a supplement to $\mathcal{M}_A$ is required, this can be expressed this with a design model $\mathcal{M}_D$.[3] We say that $\mathcal{M}_D$ *conforms* to $\mathcal{M}_A$ iff any path through $\mathcal{M}_D$ restricted to elements of $\mathcal{M}_A$ describes a path through $\mathcal{M}_A$. The set of denied operations must always be a superset of the corresponding set in $\mathcal{M}_A$. Thus, it is allowed to add edges and nodes, but not allowed to bypass elements of $\mathcal{M}_A$. In our example, the `CopyFile2Socket` concern (Figure 2) enhances the `HandleFile` concern in two ways; first it does not allow for opening a file for writing that is already open and second it adds rules for buffered copying of the data.

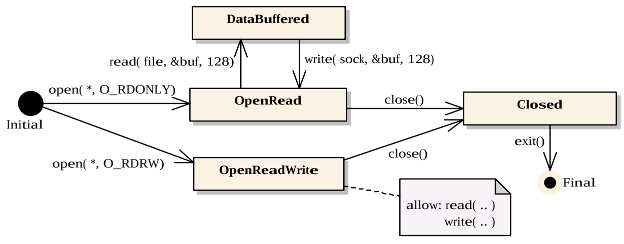

Figure 2: Design Model `CopyFile2Socket` for copying a file.

A model of the implementation can be a state machine similar to $\mathcal{M}_D$ extracted from code by means static analysis, if the set of relevant operations is known. In our approach, we do not make use of such models, but we will come back to them in the related work section. We apply Perspective Testing to system software, which needs to be reliable, such as device drivers and embedded applications. Such code is often interspersed with technical concerns that are subject to temporal constraints and usage policies. However, there is nothing domain-specific in Perspective Testing.

[3]Note that we would not call $\mathcal{M}_D$ a *refinement* of $\mathcal{M}_A$ since we can add functionality.

2.1 Example: Protocols and Policies for an Embedded Webserver

For giving an impression of the interplay of different concerns, we outline our demonstrator example, which is a working HTTP server component. It is small (few 100 LOC) but covers the important aspects of our approach. We regard the component as a whole, no matter how it is divided into classes or files, methods or functions. It is supposed to be deployed amongst other components on a embedded device with actors and sensors. Here are some candidates for concerns:

- `OpenNetwork`: Set up a socket, initialise it, bind it to an IPv4 or IPv6 address (depending on the available interfaces), accept connections and fine-tune them, close connections and sockets.
- `HandleRequest`: Fork new threads, authenticate non-local users, check and serve their requests.
- `ApplicationProtocol`: Implement higher application logic (e.g., handling sensors or actors) while serving the request.
- `LockOrder`: Respect the global locking protocol.

Furthermore, models for `CopyFile2Socket`, `HandleFile` or `AuthenticateUser` concerns can be declared. These are only examples, it is up to the engineer to choose an appropriate level and granularity of each modeled behaviour. They can overlap or even comprise each other. The range is from small and simple concerns like `HandleFile` up to large models with the complete design of outer control structures.

Depending on the actual realisation, it can be very extensive to achieve k-path coverage for the whole component. However, such a strong coverage criterion is not required: We easily identify a number of concerns, that can be tested more or less independently, for example there is no need for double-checking the whole webserver code for both IPv4 *and* IPv6 connections. For locking or authentication concerns in contrast, path covering the (related) code with tests is strongly recommended. Note that both concerns are difficult to modularise and (in our case) intermingled with the code for `OpenNetwork`.

If we take a closer look at Figure 2, we see that there is a write operation (`write(sock, &buf, 128)`) mentioned. We can suppose it to lie within the scope of the `OpenNetwork` concern. Thus, the `OpenNetwork` and the `CopyFile2Socket` concern are *not* independent from each other. It is essential that we don't require *separately modeled* concerns under test to be in fact *independent*. In this case, two concerns `HandleFile` and `OpenNetwork` are interconnected through intersecting sets of relevant operations.

3. MODEL-BASED CODE COVERAGE

Many approaches to model-based testing comprise the possibility to measure *structural model coverage*, that is counting how much of the model is covered. If the models describe the tests themselves (as in [21]), demanding 100% test-coverage is the normal case. If the model describes the environment in form of operating conditions (ranges for input data, typical usage scenarios), covering the model with tests is reasonable too. Measuring test coverage on *graphs*, such as the *class specification implementation graphs* in [6] absolutely makes sense. Opposed to that, models as described in Section 2 abstract from the concrete realisation of an concern. For example, this means that *one* path through the model may correspond to *many* paths through the code. We see the use of structural model coverage (often referred as *grey box testing*) limited to

- Providing confidence in the accuracy of the model and
- Testing, whether modeled races through the code are possible.

Although grey box testing is quite popular and widely used (since model-based testing is in vogue), we think that tests with structural model coverage are not suited for substantiating confidence in the absence of errors and thus, for complementing functional tests in the sense of our introduction.

With PERSPECTIVE TESTING we introduce the idea of defining *code* coverage criteria using models. That is, instead of covering one path through the model, we require the tests to cover many paths through the code; this is a pure white-box testing approach with models. We do this by defining an abstractor function on paths that defines which execution paths through the code (in the usual sense) are to be regarded as equivalent:

DEFINITION 1. *Two paths p and p' are* ***equivalent regarding a concern*** $\mathcal{M}_D$ *($p \equiv|_{\mathcal{M}_D} p'$), if they pass through the same relevant operations of $\mathcal{M}_D$ in the same order.*

For example, if a method closes a file on all return paths (with exactly the same operation), these paths through the method can still be equivalent regarding the `HandleFile` concern.

Our definition implies an abstraction of the control flow graph (CFG) with regard to $\mathcal{M}_D$ as a graphical representation of the equivalence classes. Any sub-graphs of the original CFG that do only represent pairwise equivalent paths are united. *It allows us to map any arbitrary coverage criterion to* PERSPECTIVE TESTING by applying it only to the concerned slice of the original CFG.

The effect on statement or branch coverage is that with an increasing number of modeled concerns, it coincides with a global branch coverage. If not, this could be an evidence for dead code or incomplete modeling. A more interesting criterion is the k-path coverage, since its complexity rises exponentially with the size of the program. This is why testing with that criterion is practically limited to unit tests (i.e. methods). By *factorising* the testing requirement along different concerns, path coverage becomes applicable for larger pieces of software, no matter how the implementation of the concern is scattered over methods and classes – we can regard the component as a whole.

Time and effort for testing two concerns with individual k-path coverage sums (in the worst case) up to $O(n+m)$ test cases instead of requiring $O(n \cdot m)$ test cases with conventional path coverage. Thus we substantially save complexity at a similar (expected) error detection rate. Another advantage of our approach is that *different coverage criteria can be combined.* In this manner, *adequate* coverage can be realised. That means, if some concerns can reasonably be tested under statement or branch coverage criteria, they can be combined with stronger criteria for more important concerns.

4. GENERATING BUILT-IN TESTS FROM MODELS

The second advantage that we take from models, apart measuring test coverage, is automatically generating state-based built-in test code for checking order constraints between operations. We require that every occurrence of a relevant operation must be "authorised" by each corresponding model. This means that the models contain information about order constraints over the relevant operations (see Section 2). We can regard the models as *behavioural contracts* that have to be fulfilled by the implementation. This can be monitored (or even enforced) by built-in tests that are automatically generated from the models and woven into the code. The scope of a BIT can be global (singleton) or be associated with a specific object or with a specific thread of control (or both). Then we need multiple instantiation of the corresponding state monitors, which keep track of the actual state of the object or thread.

4.1 Data Related Concerns

Both the `HandleFile` and the `OpenNetwork` concerns in our example are identified with data. In this case, it is a file descriptor, that is an integer representing an operating system object. Of course, during execution, there can be more than one open file or network connection; especially in object-oriented implementations these multiple instances of the same concern are probably handled by the same lines of code. Thus, a data(-flow) related observation makes sense in addition to path coverage. We instantiate a state monitor for each object when the first trigger (i.e., trigger at the outgoing edges from `Init`-states) matches. This can be a constructor call or, as in the example, a normal function call or method. The monitor is destroyed when the `Final`-state is reached.

4.2 Concurrency Concerns

In a multithreaded environment we often have to respect protocols for organising cooperation and concurrency (in addition to the functional requirements). Threads are dedicated to perform certain tasks (like producing or consuming data) and have to respect the associated rules. A prominent example for a thread-wise concern is locking. To avoid deadlocks in our webserver, one could model a global lock order that has to be respected by every thread of control. In this case, the corresponding state monitors must be associated with the current thread. It is directly possible for threads to concurrently work on one task or to hand-over responsibilities. An example is a worker-thread that has to close a network connection that was accepted by another thread.

4.3 Debugging with BIT and Filtered Traces

Conventional testing requires an observable manifestation of an erroneous race in in order to reveal any errors. This can be a faulty output or complete failure like a crash or a deadlock. This does not necessarily happen. If it does, it is often difficult to locate the reason for the failure.

The monitoring of contract violations reveals the errors in the code (bugs) directly or at least locates the errors better. If a violation is monitored, we can output a trace that provides points only to the relevant lines of code. This can be very helpful for debugging. For example if a lock order is violated, we get the history of `lock()` and `unlock()` operations of that thread. Especially lock order violations only seldom become manifest in deadlocks, such errors are hard to reveal without BIT.

5. RELATED WORK

Literature on structural testing (with coverage criteria) and model-based testing is very rich, we can not hope to cover it here. For getting started with model-based testing, we recommend the "Taxonomy of model-based testing" [26], which is quite self-contained. Regarding structural testing, we exemplarily recommend [19] for data flow coverage, [5] for control flow coverage or [18] for condition coverage.

5.1 Testing Based on Finite State Machines

Of course, our idea is not *completely* new. Especially testing based on finite state machines is a long-standing research topic, starting in 1956 with [17]. As described in Section 2, our models $\mathcal{M}_A$ may contain behaviour that is actually not realised by the software. For example, $\mathcal{M}_A$ can contain trigger that do not match any line of code of the system under test. On the other hand, the code can implement additional behaviour that is not contained in $\mathcal{M}_A$ or in $\mathcal{M}_D$ because it is not relevant. This opposes our approach to many testing approaches based on state machines, such as the *class state machines* described in [14] (picked up by [6]).

5.2 Select Relatives

Researchers from Microsoft have developed a *static* verification tool SDV [3] for signing drivers. Although it is a static approach, it is closely related to ours: In a first step, an model is extracted from code by means of static analysis. In a second step, a model checker is used for verifying temporal logic properties. The used temporal logic formulae correspond to our models $\mathcal{M}_A$ introduced in Section 2. The extracted models reflect the actually realised behaviour and are thus from the type of implementation models. Currently, the tool provides about 65 rules that describe device driver interfaces.

We found some relatives in the area of integration testing. An approach that uses automata for interface specification is described in [13]. The automata are used for automatically checking the compatibility between interface models. In [15], interfaces are specified via context free *grammars* and then checked with a model checker. Specialised coverage measures for integration testing are given in [24]; they typically use coverage criteria defined by the set of imported operations (and thus regard only a slice of the code).

Built-in self tests are recognised as useful for a long time, but are not really established. Documenting and testing method pre- and postconditions (*design by contract*) or loop invariants with assertions is rarely done. [7] is an approach that resembles ours by weaving BITs as aspects into (aspectual) components. One concrete BIT that we are interested in is the FreeBSD *Witness* [9] that surveys the FreeBSD locking protocol at runtime.

5.3 Novelty of the Perspective Testing Approach

We see the particular novelty of our approach in the idea of factorising the complexity of a software component using several (intersecting) state machines for testing purpose. If we imagine a software component as one large state machine, the same information can be expressed by a number of smaller state machines where the state of the whole is the product of the states of all smaller state machines. To our knowledge new is also the idea to use models for defining coverage criteria *on the code* by projecting model entities onto it.[4]

6. PRACTICABILITY ISSUES AND EVALUATION PLAN

We try PERSPECTIVE TESTING out with a prototypical implementation and our demonstrator example outlined in 2.1. Most parts of the implementation are straight forward, but range from meta modeling with EMF (for defining the "vocabulary" of our models) down to dealing with function pointers in plain C (used for realising polymorphic trigger implementation in C). In our prototypical implementation, we are directly using AspectC++ [25] for instrumenting the component under test with triggers. We also plan to evaluate the source transformation tool TXL [12] for instrumenting the code.[5] While instrumenting the code, we give each instrumented line a number for quick look-up in a static array of trigger-candidates. This ad hoc allows for measuring statement coverage and comparing the software behaviour with the models.

Measuring the path coverage for larger object-oriented components can become challenging. We have not much expertise in that field at our chair. To a certain extent, this will be feasible analysing the call graph using the LLVM framework (also regarding v-tables) and then building an interprocedural CFG using its relevant part. The sticking point is doing static reachability analysis.

6.1 Evaluation Plan

Our demonstrator example is a necessary first step for proving the concept and for identifying the challenges. It is barely small enough for achieving global k-path coverage for being able to comparing the error detection rates of conventional and individual path coverage. It will be interesting to investigate and classify the errors that were missed out by PERSPECTIVE TESTING.

An ambitious goal is to express the FreeBSD locking rules[6] with models and to instrument the kernel with according BIT code. This is an example with few larger models. Differentiated coverage measures will allow us for investigating and quantifying the strengths and weaknesses of the FreeBSD stress test suite [10] with regard to that concern.

A further possibility for evaluating PERSPECTIVE TESTING lies in translating the 65 API usage rules from the Microsoft SDV [2] into models. This would allow for directly comparing static and dynamic approaches and be an example with many small models.

[4]Currently our approach requires concrete operations in the models. In general, only (*traceability*) from model entities to code entities is required.

[5]TXL is a very powerful code transformation tool, which exists for decades but seems to be rarely used.

[6]Although FreeBSD is written in C, we are faced with a polymorphic implementation. Especially `lock` and `unlock` functions are called through function pointers. See `struct lock_class` in [11]. This file also describes some of the locking rules.

7. DISCUSSION

If the models are also used in the design process, we lose some of the required orthogonality between test and design. One has to be aware of the fact that models and code can contain the same errors; therefore, the models must be validated carefully (with formal methods in the best case).

We have to be are aware that controlling coverage criteria in the proposed way can be Janus-faced: Purely syntactical coverage criteria are blind for the intentions of the developer and are immune against errors in reasoning. The special value of such tests that comes from their pure objectiveness is lost. If, for example, a relevant operation has been added to a component by an API upgrade and the model was not adapted, a good coverage result can be trappy. The same can happen due to any other mistake by the tester.

One problem of strong structural coverage criteria is knowing how to valuate them. What does it mean to achieve 80% path coverage—is that enough? This highly depends on the software design; that is, its coding style and the number of infeasible execution paths (i.e., for which no input data exist). We will consider this issue in the future work section.

8. CONCLUSION AND FUTURE WORK

With PERSPECTIVE TESTING, we introduce a way of *factorising* structural tests along concerns instead of separating them along units. This avoids the exponential complexity of testing in the common case that the concerns are not perfectly modularised into one unit each and path coverage is wanted. Arbitrary coverage criteria can be applied to a sliced version of the control flow graph and concern-wise different criteria can be combined to one coverage criterion tailored to the component. The concerns are declared by means of state machines that also serve as behavioural contracts. Automatically generated built-in tests survey the order constraints derived from these models.

It is a very promising idea to combine BIT and the knowledge about illegal sequences of operations with methods of *symbolic execution* [8] for directly searching test cases that violate the modeled constraints. Symbolic execution can also be used for showing (i.e., *proving*) that a concrete path is not feasible (by disclosing the accumulated symbolic terms to be contradictory). This addresses the problem that informative value of a coverage criterion is damaged if too many unfeasible entities are contained. However, important concerns should be realised in a transparent manner that allows for deciding whether a path is feasible or not. For essential (e.g., safety critical) concerns, it can make sense to enjoin the developers on making *full* coverage possible by keeping the implementation simple enough. This requires the corresponding slice of code not to be involved in loops and means demanding an improved design and implementation for testability.

One of the current challenges in computer science is writing reliable software for the upcoming manycore platforms—and testing it. One way of systematically testing particular interleavings of threads is to integrate the state observer functionality from PERSPECTIVE TESTING in an operating system. The scheduler could then use the knowledge about the current state of the single threads for setting up the required interleaving.

9. REFERENCES

[1] S. Apel, T. Leich, and G. Saake. Aspectual mixin layers: Aspects and features in concert. In *In Proc. of Intl. Conf. on Software Engineering*, pages 122–131. ACM Press, 2006.

[2] T. Ball, B. Cook, V. Levin, and S. K. Rajamani. SLAM and static driver verifier: Technology transfer of formal methods inside microsoft. In *Integrated Formal Methods*, volume 2999 of *Lecture Notes in Computer Science*, pages 1–20. Springer Berlin / Heidelberg, 2004.

[3] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. *SIGPLAN Not.*, 37(1):1–3, 2002.

[4] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 187–197, Washington, DC, USA, 2003. IEEE Computer Society.

[5] B. Beizer. *Software testing techniques (2nd ed.).* Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[6] S. Beydeda and V. Gruhn. Integrating white- and black-box techniques for class-level testing object-oriented prototypes. In *In SEA Software Engineering and Applications Conference (Las Vegas*, pages 23–28. IASTED/ACTA Press, 2001.

[7] J.-M. Bruel, J. Araujo, A. Moreira, and A. Royer. Using aspects to develop built-in tests for components. In *In AOSD Modeling with UML Workshop, 6th International Conference on the Unified Modeling Language (UML*, 2003.

[8] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224, 2008.

[9] FreeBSD Community. The witness lock validation facility (online). http://www.freebsd.org/cgi/man.cgi ?query=witness, 2001.

[10] FreeBSD Community. The FreeBSD kernel test suite (online). http://people.freebsd.org/ pho/stress/index.html, 2008.

[11] FreeBSD Community. The FreeBSD source code version 8.0 (online). http://svn.freebsd.org/base/stable/8/sys/sys/lock.h, 2009.

[12] J. R. Cordy. Source transformation, analysis and generation in txl. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 1–11, New York, NY, USA, 2006. ACM.

[13] L. de Alfaro and T. A. Henzinger. Interface automata. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 109–120, New York, NY, USA, 2001. ACM.

[14] H. S. Hong, Y. R. Kwon, and S. D. Cha. Testing of object-oriented programs based on finite state

machines. In *APSEC '95: Proceedings of the Second Asia Pacific Software Engineering Conference*, page 234, Washington, DC, USA, 1995. IEEE Computer Society.

[15] G. Hughes and T. Bultan. Interface grammars for modular software model checking. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 39–49, New York, NY, USA, 2007. ACM.

[16] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *11th (ECOOP '97)*, volume 1241, pages 220–242, June 1997.

[17] E. F. Moore. Gedanken experiments on sequential machines. In *Automata Studies*, pages 129–153. Princeton U., 1956.

[18] G. J. Myers and C. Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.

[19] S. Rapps and E. J. Weyuker. Data flow analysis techniques for test data selection. In *ICSE '82: Proceedings of the 6th international conference on Software engineering*, pages 272–278, Los Alamitos, CA, USA, 1982. IEEE Computer Society Press.

[20] J. M. B. Roger T. Alexander and A. A. Andrews. Towards the systematic testing of aspect-oriented programs. *Technical Report CS-4-105*, 2004.

[21] B. Rumpe. Model-based testing of object-oriented systems. In *In: Formal Methods for Components and Objects, International Symposium, FMCO 2002, Leiden. LNCS 2852*. Springer Verlag, 2003.

[22] F. Saglietti, N. Oster, and F. Pinte. Interface coverage criteria supporting model-based integration testing. In Marco Platzner, Karl-Erwin Großpietsch, Christian Hochberger, and Andreas Koch, editors, *ARCS '07 - Workshop Proceedings*, pages 85–93, Zürich, 2007. VDE Verlag GmbH Berlin/Offenbach.

[23] D. Sokenou and S. Herrmann. Aspects for testing aspects. *Workshop on Testing Aspect-Oriented Programs AOSD05*, 2005.

[24] A. Spillner. Test criteria and coverage measures for software integration testing. *Software Quality Journal*, 4(4):275–286, December 1995.

[25] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. Aspectc++: An aspect-oriented extension to c++. In *In Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002*, pages 53–60, 2002.

[26] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing. Technical report, University of Waikato, April 2006.

[27] W. Xu and D. Xu. State-based testing of integration aspects. In *WTAOP '06: Proceedings of the 2nd workshop on Testing aspect-oriented programs*, pages 7–14, New York, NY, USA, 2006. acm.

[28] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.