Achieving Both Model and Code Coverage with Automated Gray-Box Testing

Nicolas Kicillof
FCEyN - UBA
nicok@dc.uba.ar

Wolfgang Grieskamp,
Nikolai Tillmann
Microsoft Research
{wrwg,nikolait}@microsoft.com

Victor Braberman *
FCEyN - UBA / CONICET
vbraber@dc.uba.ar

ABSTRACT

We have devised a novel technique to automatically generate test cases for a software system, combining black-box model-based testing with white-box parameterized unit testing. The former provides general guidance for the structure of the tests in the form of test sequences, as well as the oracle to check for conformance of an application under test with respect to a behavioral model. The latter finds a set of concrete parameter values that maximize code coverage using symbolic analysis. By applying these techniques together, we can produce test definitions (expressed as code to be run in a test management framework) that exercise all selected paths in the model, while also covering code branches specific to the implementation. These results cannot be obtained from any of the individual approaches alone, as the model cannot predict what values are significant to a particular implementation, while parameterized unit testing requires manually written test sequences and correctness validations. We provide tool support, integrated into our model-based testing tool.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.5 [**Software Engineering**]: Testing and Debugging—*Symbolic execution, Testing tools*; D.2.10 [**Software Engineering**]: Design—*Representation*

General Terms

Algorithms, Design Verification

Keywords

Model-based testing, parameterized unit testing, concolic execution, symbolic execution, test-case generation

1. INTRODUCTION

At Microsoft Research, we have in recent years developed a tool environment for model-based testing called Spec Explorer [6]. This environment allows users to model object-oriented, reactive software, analyze the specification with model-checking techniques and automatically derive model-based tests. Its current version [10] is integrated into a development environment and supports advanced features such as *notational independence*, *multi-paradigmatic modeling* and *model composition*. Our research group has also developed a new approach to white-box testing, parameterized unit testing [20], implemented in a tool called Pex. Parameterized unit tests (PUTs) are code fragments, usually consisting of sequences of method invocations to the unit under test, combined with simple control-flow structures. They also allow users to express specifications of expected behavior using assertions. Contrary to a regular unit test, each PUT has one or more parameters (free program variables). Pex automatically selects test inputs for the parameters in an iterative process, implementing a variation of symbolic execution [9]. Starting with arbitrary inputs, Pex monitors the execution of the software system and characterizes the set of inputs which are likely to result in the same execution path. A constraint solver automatically determines new, not yet characterized inputs. The process stops when all inputs have been characterized, or configurable bounds are exceeded. At this point, Pex outputs as its result a set of unit tests: instantiations of the PUTs with a set of values that maximize coverage criteria.

This article reports on the combination of both techniques. It was motivated by feedback from product groups at Microsoft and external enterprise customers. In early presentations of Spec Explorer users have shown their enthusiasm for being able to write their models in various notations, to combine them in different ways, to check them against expected properties and to automatically generate test cases. Nevertheless they were concerned about the fact that the tool does not guarantee to achieve high code coverage, one of their main measurements of test quality. As to Pex, users greatly appreciate its power to find relevant values based on code itself, and hence its increased code coverage with respect to both random and manual testing. But Pex still requires manual writing of (parameterized) tests with embedded oracles, and it does not provide guidance on which tests to write or how many of them are needed. Also, it is not always clear whether the test oracle should be expressed as an expected behavior in a PUT or whether it should be an assertion in the application code. As we show in our running example, developers can regularly adjust application assertions to match code changes, which maintains these two synchronized, but violates requirements reflected by design documents.

As a response to these shortcomings, we have decided to build a combination of both tools. In this novel approach, Spec Explorer is used to interactively write and check a rich-state model that conveys user requirements in the form of valid execution sequences and constraints on values and data structures. It then allows users

*Supported by UBACyT X020, and PICT 11738 and 32440

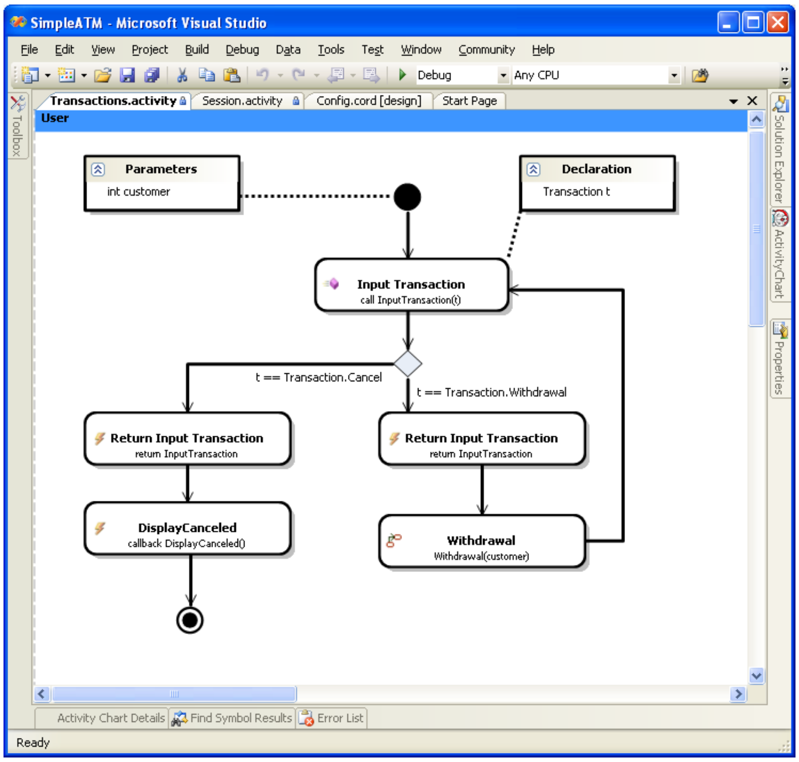

Figure 1: Activity diagram for use case "Transactions"

to apply various exploration techniques in order to generate PUTs that encode both significant scenarios and their expected outcomes. These PUTs are still black-box in the sense that they have been generated from the model, without considering a specific implementation. A set of concrete values can be provided to instantiate certain parameters while others are left symbolic. Then Pex explores the PUTs in order to find values for the latter that are relevant to the application code. The results are tests that encode test purposes and oracles provided by Spec Explorer, fully instantiated with the values found by Pex, which result in high code coverage.

This paper is organized as follows. Sect. 2 briefly describes our previous tools and lessons learned in applying MBT at Microsoft. Sect. 3 presents our running example and the problems it poses for automatic test generation. Sect. 4 explain how our proposed approach solves these problems. Sect. 5 and 6 conclude.

2. OUR TOOLS

Model-Based testing with Spec Explorer

Testing is one of the most cost-intensive activities in the industrial software development process. Yet, not only is current testing practice laborious and expensive, but often also unsystematic, lacking engineering methodology and discipline, as well as adequate tool support. Model-based testing (MBT) is one of the most promising approaches to address these problems. At Microsoft, MBT technology has been applied in the production cycle since 1999 [17, 11, 3, 19, 6, 10]. One key for the relative success of MBT at Microsoft is its attraction for a certain class of well-educated, ambitious test engineers, who see it as a way to raise testing to the level of a systematic engineering discipline.

However, from a broader perspective, we estimate from the number of subscriptions to Microsoft internal mailing lists for MBT that only about 5-10% of product teams have used or tried MBT in their daily tasks. While these numbers can be considered a success compared to other formal quality-assurance approaches like verification, they are certainly not indicating a breakthrough. In recent years, we have analyzed the major obstacles to applying MBT, and extended our tools with features that we trust will attract a larger group of users to the technology [10].

Our conclusions are based on feedback from the user base of the first version of Spec Explorer [6], its predecessor AsmL-T [3], and other internal MBT tools at Microsoft. The main issues (apart from the ubiquitous problem in industry that people do not have enough time to try out new technology and get the required training) seem to be the steep learning curve for modeling notations together with the lack of state-of-the-art authoring environments, missing support for scenario-based modeling (which could help get not only the test organization but also other stakeholders involved in the process), poor documentation of MBT tools; and, last but not least, technical problems like dealing with state explosion, fine-grained test selection, and integration with test management tools.

Our new model-based testing system is currently under development at Microsoft Research and it attempts to overcome some of these obstacles. The tool, called "Spec Explorer 2007", tries to address the identified challenges by providing full integration into the Visual Studio development environment, using a *multi-paradigmatic* approach to modeling. It allows the user to describe models on different levels of abstraction using scenario and state oriented paradigms as well as diagrammatic and programmatic notations, and it enables the combination of these diverse artifacts for a given modeling and testing problem.

Spec Explorer 2007 is internally based on the framework of action machines [13, 12], which permits uniform encoding of models stemming from a variety of notations, and their combination using

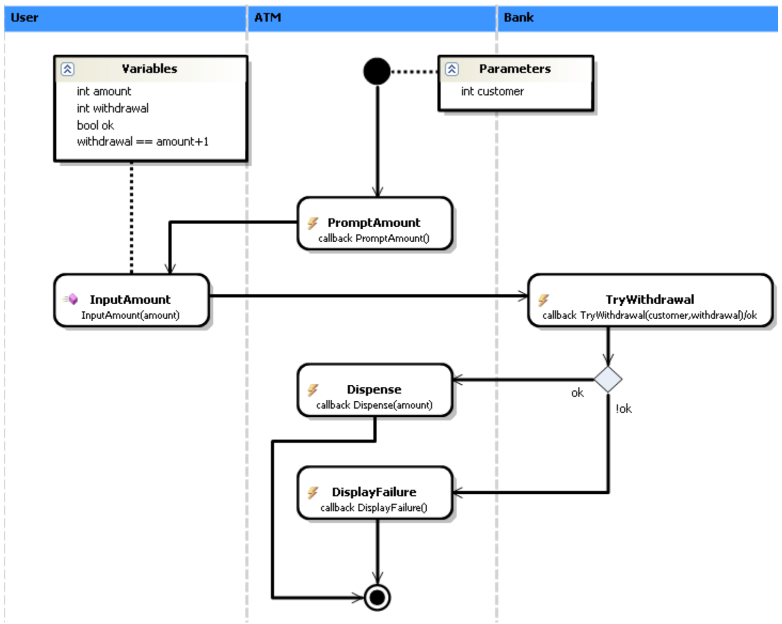

Figure 2: Activity diagram for use case "Withdrawal"

different composition operators. The action machine framework supports models with symbolic representation for parts of states and actions, which gives rise to the expressive power of defining partial models on a high level of abstraction and composing them with lower-level models.

Parameterized Unit Testing with Pex

Unit tests are becoming increasingly popular. According to a recent survey at Microsoft, they are being applied by 79% of the developers [24]. Their purpose is to document requirements, to reflect design decisions, to protect against changes but also, as part of the testing process, to achieve certain code coverage, which leads to a high confidence in the correctness of the application.

The growing adoption of unit testing is partly due to the popularity of methods like Extreme Programming (XP) [4] and test execution frameworks like JUnit [8]. XP promotes Test-Driven Development (TDD) [5], where unit tests are written to guide feature implementation. However, these unit tests usually cover only specific cases, and XP does not provide a way to determine when enough tests have been written. Test execution frameworks automate only test execution, but they do not automate the task of creating a comprehensive set of unit tests. Writing all unit tests by hand can be a laborious undertaking. In many projects at Microsoft unit tests take more lines of code than the implementations being tested.

Although first envisioned in 1976 [15], symbolic execution has only recently become feasible in practice. This is due to improvements in hardware and the development of better algorithms for automatic reasoning. A breakthough for test-case generation is the under-approximation of the full behavior of the application by the process of monitoring concrete application executions, combined with using constraint-solving techniques to obtain new test inputs [9]. Both traditional testing and TDD benefit from these techniques because test inputs -including the behavior of entire classes– can often be generated automatically from compact PUTs.

Pex monitors the execution of .NET programs by instrumenting the program's instructions. It inserts callbacks which allow the precise monitoring of the program's control- and data-flow. During the analysis of an execution path, Pex computes a symbolic state representation. At any time, it represents the current state of the program using terms over the program's input. Terms represent computations already performed by the program. Also, at every conditional branch, Pex records the guarding term of the branch condition. The conjunction of all branch conditions in an execution path is termed *path condition*. It characterizes the set of inputs which are likely to result in the same execution path. (We say 'likely' since the program may perform operations which Pex does not understand or is not able to represent as a term, such as network communications.) At this point, Pex uses a custom constraint solver to find inputs which do not satisfy any previous path condition. Pex's constraint solver implements decision procedures for logical constraints, equalities and inequalities over terms, a subset of linear arithmetic, and arrays. Pex also attempts to find solutions within predefined finite domains.

3. CASE STUDY

Our running example is a variation of the well-known example of an automatic teller machine (ATM) control system. For the purpose of this article, we will show only one of the modeling notations supported by Spec Explorer 2007: extended activity diagrams, to describe the expected behavior of the ATM in an interaction-based style.

Our model consists of three hierarchically organized use cases. The top-level one, "Session", describes the whole interaction of a customer with the bank via the ATM system, from the time a card is inserted until it is ejected; use case "Transactions" describes iterating transactions as part of this interaction; and use case "Withdrawal" describes an individual transaction.

Fig. 1 shows a screen shot of Spec Explorer 2007 (within Visual Studio) displaying the activity diagram for use case "Transactions". It portraits a loop where the user enters a transaction type (variable t). Depending on it, either the "Withdrawal" use case is invoked, or processing of further transactions is cancelled (this is withdrawal-only ATM, for simplicity sake).

Such scenario descriptions might result from the requirements

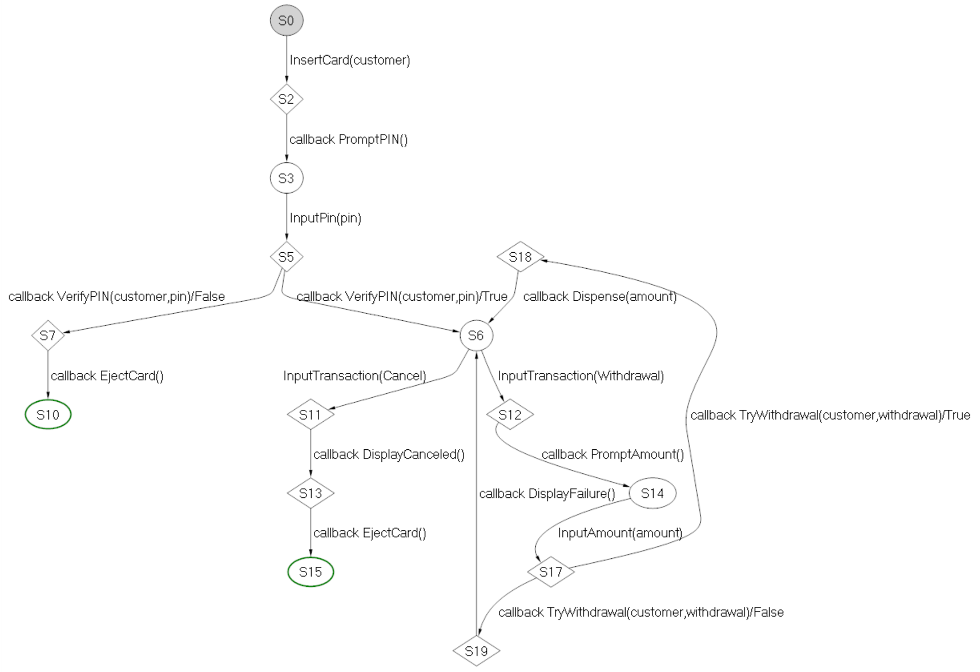
S0
InsertCard(customer)
S2
callback PromptPIN()
S3
InputPin(pin)
S5
S18
callback VerifyPIN(customer,pin)/False
callback VerifyPIN(customer,pin)/True
callback Dispense(amount)
S7
S6
callback EjectCard()
InputTransaction(Cancel)
InputTransaction(Withdrawal)
S10
S11
S12
callback TryWithdrawal(customer,withdrawal)/True
callback DisplayCanceled()
callback PromptAmount()
S13
callback DisplayFailure()
S14
callback EjectCard()
InputAmount(amount)
S15
S17
callback TryWithdrawal(customer,withdrawal)/False
S19

Figure 3: Exploration graph for scenario "Session"

phase or from modeling test plans. Initially, nodes are typically labeled with abstract names, such as "Input Transaction". Later on, in the course of making the model more concrete for analysis or testing tasks, nodes are mapped to *action patterns*. In the screenshot, the text underneath the title "Input Transaction" shows such a mapping. Namely, this activity is mapped to the action invocation call InputTransaction(t). In this case, the mapping was performed manually, based on an underlying object model for the ATM (obtained by reflection from the implementation, once it became available); but Spec Explorer can also perform it automatically by applying certain heuristics [1].

We support three kinds of activity nodes in our version of activity diagrams. Activity nodes identified with a flying box (like "Input Transaction" in Fig. 1) represent operations that can be externally called on the system under test (SUT); we term them *actions*. Activity nodes marked with a flash (like "DisplayCanceled') are actions trigger by the SUT itself, which can be observed from the outside (as events, callbacks, or returns from controllable actions); we call them *signals*. Finally, activity nodes with a diagram symbol (like "Withdrawal") are placeholders for hierarchically composed behaviors specified elsewhere (in any of the different notations supported by Spec Explorer).

Variables in activity diagrams play an important role for the expressiveness of the approach: they correlate inputs and outputs from different activity nodes. All variables are purely declarative. Constraints over variables can be added as labels to arrows (flows). Variable scope can be limited to a subgraph of an activity, like variable t in Fig. 1, which has the transaction loop as its scope and can hence represent a different value in each iteration. The static scope of this variable is expressed in the notation by attaching its declaration block to the first node in the loop, according to the reachability relation from the start node. The variable will be considered fresh every time exploration hits the declaration node.

[1]*Translations* can also be applied in order to map model actions to different implementations.

Use case "Withdrawal" (Fig. 2) is slightly more complex. It begins with the ATM prompting for an amount ("PromptAmount"), which is then entered to the system by the user ("InputAmount"). The ATM then requests the bank to attempt a withdrawal from the user's account by invoking operation "TryWithdrawal" whose result will be stored in variable ok. If this operation is successful, the ATM will dispense the requested money ("Dispense"); otherwise, it will inform the failure to the user ("DisplayFailure"). The correlation between the arguments to all involved activities is again realized by using (logical) variables. In particular, variable amount insures that the amount input by the user in "InputAmount" is the same as that dispensed by the ATM in "Dispense".

Besides showing valid interactions that can occur during system execution, the chart in Fig. 2 also conveys one important requirement restricting data exchanged in such interactions. The second argument to "TryWithdrawal" is variable withdrawal, which, per a constraint stated in the "Variables" block (top left corner), is 1 more than the amount input by the user. This reflects the fact that, in an implementation conforming to this model, the ATM must charge a $1 fee for every withdrawal transaction.

Exploring the ATM Model

Once this model is drawn in Visual Studio, we can begin *exploring* it under Spec Explorer. A simple exploration of the "Session" scenario yields the graph in Fig. 3. It basically depicts a labeled transition system (LTS) with *interface automaton* [7] semantics. Its nodes represent states and its transitions represent action invocations. Circle nodes are control points where input is provided to the system by invoking one of its operations from the outside. Diamond nodes are observation points where the system is observed to invoke an operation itself. Note how variable amount, on the right-hand side (representing the amount entered by the user), maintains the causality expressed in the model, which goes beyond pure control flow: the same amount input in InputAmount(amount) must also be dispensed by the ATM (callback Dispense(amount)). If Spec Explorer is asked to expand the assumptions contained in state S18 (by double-clicking on it), it shows constraint 1+amount == withdrawal, which represents (still in symbolic form) the charging of a fee required by the model.

Although it constitutes an expansion of the model, exploration still maintains two kinds of partiality. On the one hand, the value of variables such as amount is not fixed. On the other hand, the model does not provide any information on how to determine the success of a PIN-verification operation (state S5) or a withdrawal from the bank (state S17); it only states what the successive behavior is supposed to be for each of the possible outcomes of these operations.

From this point on, the model composition features of Spec Explorer 2007 permit manipulating the model in several ways. For example, it can be combined with a model expressing the behavior of the bank, in order to reduce partiality by going into detail on how a PIN is verified, or when a withdrawal is successful and how it affects an account's balance. Composition can also be applied to model-check a specification against a property also expressed as a (partial) scenario.

Spec Explorer can automatically test an implementation for conformance to a model. For this task, the user can supply concrete sets of values for the logical variables referenced in a model. A simple way to do this is through the action machine coordination language, Cord [12], a declarative intermediate notation that constitutes a textual frontend for action machines. It can be used to define action patterns, compositions between behaviors, and (as in this case) configurations for model-based testing and model-checking problems, like parameter generators, exploration bounds, traversals, and so on. By writing the following Cord expression, we build a new behavior resulting from replacing symbolic parameters in the "Session" behavior with the supplied concrete values:

```
construct parameter expansion for
  bind call InsertCard(1), call InputPin(1,2), call InputAmount(9)
  in Session()
```

This behavior can be explored by itself or used to generate test cases that can be run against an implementation from inside the tool in order to validate conformance to the model. In Fig. 4 we see the results of such an online test in an implementation that fails to take the fee for withdrawal transactions. State S14 and S15 correspond to conformance failures, where the expected operation (TryWithdraw(1,10)) was not performed by the SUT. If one of these nodes is expanded, Spec Explorer shows that the observed operation was instead TryWithdraw(1,9).

The Case for Code Coverage

Let us now assume that the bug described above is fixed by making the ATM implementation charge the required fee, so that the application passes all tests automatically generated from the model by Spec Explorer. Some time later, one of the developers in charge of maintaining the application receives from a customer (an employee in one of the banks using the system) the requirement that transaction fees must not be taken from VIP clients. This particular bank considers VIP those clients with a balance of $100,000 or more. The developer is very eager to satisfy the request, and thus proceeds immediately to enclose the existing fee-charging code in a conditional structure that prevents its execution whenever the account balance is greater than $99,9999.

At a first glance, this could be considered a job well done. The developer has quickly and effectively succeeded to comply with a customer's request. Nevertheless, from the point of view of the software development process, this was not a wise decision. The model still demands that the fee must be taken in every case. And there is a good reason for this: the bank that issued the requirement is not the only one using the system. Other banks do not have VIP clients, or take regular operation fees from them. Even if this were a common practice in the financial business, the criterion to establish who is a VIP might vary from institution to institution. In case a modification must be introduced to the system to prevent taking fees in particular cases, it should only be made after a careful design decision, reflected in the model.

We therefore expect the modified implementation to fail at least one of the generated conformance tests. Unfortunately, as we have seen, the argument values in Spec Explorer-generated test cases are those relevant to the model. It is unlikely that, being unaware of the change made to the code, a test engineer, modeler or black-box automatic parameter generator will come up with a test value of $100,000 or greater. Obviously, tests with large values are meant to be part of a test suite, since overflows are frequently a source of errors in software. But in our example we have chosen the number 100,000 as a representative of a value that was not considered significant for model-based testing, as it produces no distinct behavior in the model itself. For a more extreme example, we could suppose the new rule introduced in the implementation prevents taking a fee from accounts with a balance of *exactly* $10,001 (maybe as an incentive from the bank for its small clients to deposit more money). Our example would be equally valid for this modified requirement, and would now depend on at least one test case to be generated for this specific value (not *any* large value), which is again not known from the model's point of view.

4. GRAY-BOX TEST GENERATION

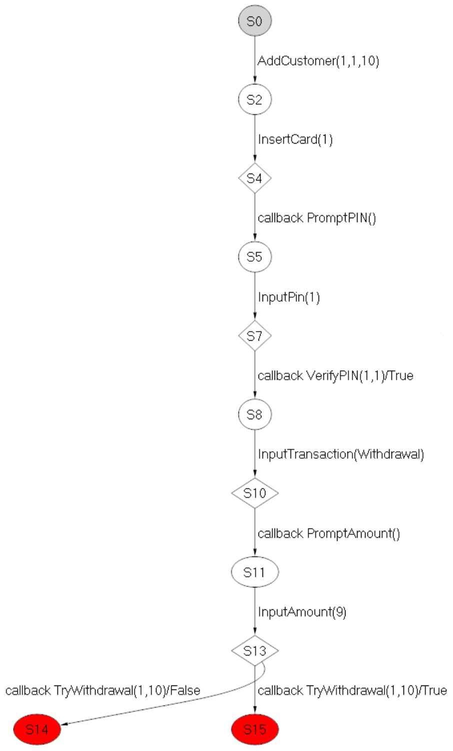

Figure 4: MBT result from Spec Explorer

Our solution to the problem posed in Sec. 3 consists in integrating the presented tools into a tool chain that combines their best features: the high-level modeling notations that serve as a source to generate black-box tests in Spec Explorer, and the ability of Pex to find values that are relevant to a specific implementation via white-box test-case generation. We call this tool chain "Automated Gray-Box Testing" (AGBT).

AGBT is launched from inside Visual Studio, by first exploring a Spec Explorer set of models and selecting an exploration result as test scenario. The key step in the choice of the source model and the exploration settings is to keep symbolic those parameters that the user wants to be generated by Pex. In our example, we will not bind the initial balance of accounts to a definite value. We thus obtain from Spec Explorer a PUT, where the initial balance is left to be instantiated by Pex. Another approach would be to leave all parameters symbolic, which would increase the time consumed by Pex and the number of generated test cases.

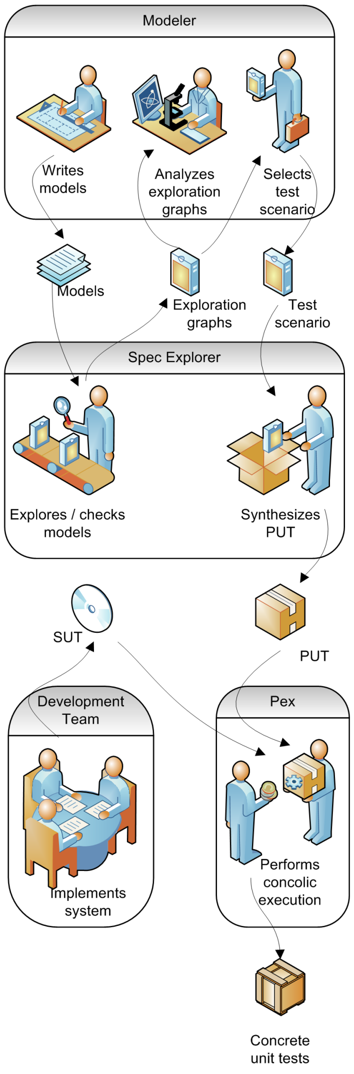

Figure 5: AGBT **Workflow**

Fig. 5 presents the workflow of AGBT. A user begins by modeling requirements, using (a combination of) notations and styles available in Spec Explorer. Models can be explored, refined and model checked until the result reflects the intended system design. Exploration consists in unfolding all behaviors of a model, applying an algorithm similar to model-checking. A model scenario is then selected (or built) as the source for test-case generation. Spec Explorer synthesizes a PUT in the form of C# code containing the test plan and the oracle. Pex performs white-box test-case generation on the PUT and the system under test. The results are concrete unit tests, each invoking the PUT with particular test inputs found by Pex to be relevant. The unit tests are emitted as code. They can be run in a test management system for bug reproduction or regression. Pex also provides information about bugs found and coverage measurements.

As explained, in our running example we have chosen to keep the initial balance symbolic at account-creation time. Spec Explorer hence generates test code which will force Pex to generate this value at runtime. Then, this code is compiled and the resulting assembly is provided as an input to Pex, together with the (faulty) implementation. After running for less than 1 minute, Pex generates 5 test cases, with the following values for the initial account balance: $\{0, -1, 100000, 10, -1\}$, achieving a 98.53% coverage of the explored assembly. The repeated value is due to different execution paths based on the transaction choice (Withdrawal or Cancel), which has also been left undefined by Spec Explorer in order to derive tests that exercise more functionality from a single scenario. As expected, test number 3 reveals the bug. Additional coverage is easily achieved by leaving more parameters symbolic, although this exceeds the goal of the case study: to detect the VIP bug.

Perhaps Pex could have detected the error without the help of Spec Explorer (in the absence of a model), provided that requirements (such as charging fees) would have been expressed as assertions, either in the implementation code or as part of manually written PUTs. Nevertheless, it can be argued that such specifications, expressed only in code, are usually maintained by the programmers themselves and thus kept synchronized with the code. For example, the developer of our example would most probably have added the VIP case to his assertions as well as to the functional code itself. Behavioral models are, on the contrary, abstract documents that present design decisions in notations readable by most stakeholders involved in the development process. A change in such a document would only be made after achieving a consensus among those stakeholders and would need to maintain high-level global properties that can be verified through model checking.

Another advantage of models over assertions is that they provide oracles over global system behavior, rather than locally for individual methods. Even if contracts are provided for every method in a system (which might be an overwhelming task for large code bases), they tend to be partial and seldom capture functional properties of the system as a whole. Even if all methods satisfy their specifications, this does not necessarily imply the fulfillment of more abstract end-to-end properties that are expressed in models.

Dealing with Choices

The output of the model exploration phase is an exploration graph encoded as an LTS. Transition labels are actions, which have already been translated into .NET methods through the translation process explained in section 3. States contain constraints that are active at the corresponding point in the model exploration. The code-generation phase takes such LTSs as input and produces code that will steer the SUT to exhibit the behavior specified in the LTS, checking conformance in each step. This code can be used either as self-contained test cases to be run by a test framework, or as PUTs to be processed by Pex. This flexibility is provided by Spec Explorer's testing runtime architecture, which supports different *test managers* as its main extension point. Test managers are software components implementing interface ITestManager. Classes PexTestManager (for Pex) and StandaloneTestManager are two such implementations; users can develop their own.

As shown in Sec. 3, the LTS resulting from exploration can still contain choices. The key challenge in the test-code-generation phase is dealing with these choices. We roughly classify them into three large groups:

- **Controllable action choices** must be made in states with two or more outgoing transitions labeled with controllable actions. These are points where the test harness must choose among different methods to invoke on the system under test (or different argument combinations for a single method). An example of a controllable action choice can be found in state S6 of Fig. 3, where method InputTransaction can be called with either Cancel or Withdraw as its argument.

- **Observable action choices** appear in states with two or more outgoing transitions labeled with observable actions. They are points where the model permits the system under test to make one in a set of admissible invocations. For example, in state S5, the implementation should be observed to call method VerifyPIN with a result of either True (valid PIN) or False (invalid PIN). We assume the model to be deterministic for this kind of choice (a given observation can have at most one matching transition in a given observation point, taking into account target-state constraints).

- **Data choices** are introduced by variables that were left symbolic in the model, for which a value must be selected at test-run time and provided to the application. In state S0, for example, method InsertCard will be called, which has an account number as its sole parameter. The value to pass as an argument has to be chosen according to some criterion.

Controllable action choices are the main mechanism to achieve *model coverage*. Although these choices can be completely eliminated by Spec Explorer during exploration by means of different traversal strategies; the user may decide not to do so, and leave controllable action choices as part of the resulting LTS in order to more compactly represent several use cases. Since we generate tests from arbitrary LTSs, we must cope with these choices as well.

When a generated test case reaches a state where a controllable action choice has to be made, it delegates this task to the test manager by invoking method **int** ITestManager.ChoosePath(**int** pathCount). The latter should guarantee some fairness, in order to cover as many paths in the model as possible.

In the AGBT setting, this fairness is realized by asking Pex to provide an integer parameter value as an additional test input. Initially, Pex will chose an arbitrary value and monitor the continuing program execution. It will record all uses of the value, and build symbolic representations of all branch conditions in the program which depend on it. When Pex's constraint solver finds that another value would cause a conditional branch to jump to a different target; Pex executes the test code again, providing the new value when asked. From Pex's point of view, the code to be covered encompasses both the SUT and the PUT.

Observable action choices are actually driven by the SUT. There are two causes for observable action choices: *model looseness* (also found in the literature as *model partiality* or *subspecification*) and *implementation non-determinism*. The former is due to the decision of a designer not to model certain aspects or details of a system's behavior. The example we have selected (state S5 of Fig. 3) is such a case: the model contains no information as to when a PIN must be considered valid. A more detailed model (probably a refinement of this one) might specify what the outcome of the verification is for each possible PIN (by consulting some storage or data structure). The other cause for observable action choices are points where the application reacts in one of several ways, depending on some internal state (or external communication) not known to the model. This results in the possibility to make different choices *for the same execution path*. That is, given the same controllable actions invoked, and the same input data provided, the application can react in different ways. In order to cope with implementation non-determinism and try to cover as much of the model as possible, we adopt the non-intrusive strategy of re-executing the application several times under the same conditions.

According to the notion of alternating refinement [1], the basis of our definition of model conformance, any action observed in the application not expected by the model must result in a conformance failure. In the context of the LTS, from which test cases are generated, this means that observed method calls must be matched to one of those in an observable action choice point. This encompasses unifying arguments in the observed invocation to those of the expected action, and also checking that all constraints in the target state hold.

This mechanism must be applied to each of the actions in an observable action choice that can match the actual action observed from the application, until one is found for which argument values unify with those in the model and all target-state constraints hold. This trial-and-error setting requires evaluating constraints in a contained environment, where variables appearing in action arguments have been assigned values that may need to be forgotten in case the transition is not completely validated.

In order to cope with potential latency problems in cases where a (loosely coupled) SUT might invoke the expected methods before the test case is ready to observe them, these calls are enqueued by the test harness. A configurable timeout determines how long an observation is expected. Once expired, the test harness considers the application is quiescent [22]. If the current state also has one or more controllable outgoing transitions, one of them is chosen according to the strategy described above, and execution continues (after a separately specified timeout). If no control step is possible, test case execution is terminated, and success or failure of the test case is decided depending whether the current state is accepting.

To this end, variables that are left symbolic after model exploration are implemented as *transactional memory* entities. When unbound variables occur in arguments of an observable action, they receive a value from the SUT in the context of a transaction that can be rolled back (if target state constraints fail). If the test case encounters a bound variable in one of these output positions, it will compare the observed value with the one contained in the variable, as part of the conformance check, thus verifying the correlation relations encoded in the model.

Data choices are the main reason for having integrated Pex in our tool chain. As a result of model exploration, Spec Explorer can only choose among values that have either been manually provided by the modeler or are relevant to increasing model coverage. The standalone test manager lives in the realm of black-box testing and can not escape this limitation. On the other hand, when an instance of PexTestManager receives the (generic) message T ITestManager.Generate⟨T⟩(), it invokes a placeholder method that will cause Pex to provide, during PUT exploration, a new value and to track all its uses. If a further control-flow is found that is conditional on this value or a value derived from it, Pex's constraint solver will attempt to compute another value which would cause a different control-flow to be taken. Pex systematically explores all conditional control-flows by re-executing the test code (and, hence, the SUT). Transactional variables effectively request Pex to provide them with a value when they must be passed as input arguments to the SUT at a point in the execution where they are still undefined.

For example, when transitioning from state S14 to state S17 of Fig. 3, a value needs to be produced for variable amount (the withdrawal amount to be entered in the system). From then on, the variable is considered to be bound to that value. In the transition from state S18 to state S16, an invocation to method Dispense is observed and the argument is compared to the value stored in amount to ensure that the amount dispensed matches the input one. Variable withdrawal, on the contrary, is bound in either of the outgoing transitions of state S17 to the value observed in the communication between the ATM and the bank. This causes the evaluation of the constraint requiring this value to be \$1 greater than amount, which has been postponed for having at least one unbound variable, as explained below. The transition from state S18 to S6 has the additional effect of *unbinding* variables amount and withdrawal, since it is a transaction that closes a loop to which variable scope is limited.

Due to the dynamic nature of variable-value binding, certain constraints may appear in a state for which not all of their variables are bound in some execution paths. Spec Explorer's exploration algorithm guarantees that these constraints appear replicated in every state where they can be effectively evaluated. This permits us to safely avoid checking constraints whose variables are not bound yet, lazily differing their validation.

An optimization of this dynamic mechanism is implemented as part of transition-system building, in the form of a static binding (definition-use) analysis. This minimizes the checks that need to be performed during test execution, which results in both better test-execution performance and generated-code readability.

Structure of Generated Test Cases

Having examined the main concerns that guide the test-case-generation process, we proceed to analyze the structure of the resulting code, shown as pseudo-code in the following listings.

```
1  [TestSuite]
2  class GeneratedTest : TestSuiteBase
3     Variable<int> amount;
4     Variable<int> balance;
5     Variable<int> customer;
6     Variable<bool> ok;
7     Variable<int> pin;
8     Variable<bool> pinOK;
9     Variable<int> withdrawal;
10
11    string currentState;
12    bool finished;
13    bool failed;
```

The test class is declared in line 2. All code contained in the listings is part of this class. Line 1 marks it as a test suite through a .NET custom attribute. This is the mechanism used by unit test frameworks (and also by Pex) to discover test suites and test cases via reflection on metadata.

Lines 3-9 declare all variables occurring either in actions or in states (constraints) of the transition system, as fields (instance variables) of the test class. Each field is statically typed during code generation using a particular instantiation of the generic type Variable⟨T⟩. This class also contains a set of control-related fields, declared in lines 11-13.

```
14 void InitializeVariables()
15    amount = new Variable<int>("amount", Manager);
16    balance = new Variable<int>("balance", Manager);
17    customer = new Variable<int>("customer", Manager);
18    ok = new Variable<bool>("ok", Manager);
19    pin = new Variable<int>("pin", Manager);
20    pinOK = new Variable<bool>("pinOK", Manager);
21    withdrawal = new Variable<int>("withdrawal", Manager);
```

Method InitializeVariables in lines 14-21 creates new instances of the corresponding Variable class and stores them in the field representing the model variable. It is invoked at the beginning of each test case execution, in order to start with fresh, unbound variables.

Variables are initialized with a string representing their name (for reporting purposes) and a reference to the test manager (from which they need to request a value when required). Property Manager is declared in class TestSuiteBase, the superclass of all test suites.

```
22 [TestCase]
23 public void TestCase0()
24    int runs = 0;
25    do
26       Manager = new PexTestManager();
27       InitializeVariables();
28       Test("S0");
29       runs++;
30    while (runs < Manager.Reruns)
31
32 [TestCase]
33 public void TestCase1()
34    ...
35 ...
36 [TestCase]
37 void TestCasem()
38    ...
```

Methods TestCase0-TestCase*m* in lines 22-38 are the actual test cases (for standalone testing) or PUTs (for Pex). Each is marked with a custom attribute (lines 22, 32, 36) to make it discoverable by either a test framework or Pex.

There is one test case per initial state in the transition system, consisting in a loop (lines 25-30) controlled by the test manager. In each iteration, all logical variables are initialized (line 27) and a test method common to all test cases (defined in line 39, below) is invoked (line 28) on the corresponding initial state. The loop simply repeats the whole process as many times as the test manager considers it fit, in order to try to cover implementation non-determinism (see section 4).

```
39 void Test(string initial)
40    string currentState = initial;
41    while (!finished)
42       switch(currentState)
43          case "S0": StateS0();
44          case "S1": StateS1();
45          ...
46          case "St": StateSt();
```

Method Test controls the flow to encode the transition system as a state machine. It starts by setting its argument as the current state (line 40). In line 41 the state machine loop begins. It contains a **switch** statement that transfers control to the method associated with the current state.

```
47 void StateS0()
48    ControllablesS0();
49
50 void ControllablesS0()
51    Manager.BeginTransaction("AddCustomer(1,1,balance) → S1");
52    ATM.AddCustomer(1,1,balance.Value);
53    Observation.AddReturn("ATM.AddCustomer");
54    if (Manager.EndTransaction())
55       currentState = "S1";
56    else finished = failed = true;
```

A simple example of such a method can be seen in lines 47-56 for state S0. This is a state with a single controllable outgoing transition labeled AddCustomer(1,1,balance), actually invoked on the SUT in line 52. The value of variable balance is queried which causes a request for a new value to be issued to the test manager, in case the variable is still unbound. Pex will interpret this request as the need to provide data. If the transition system imposes constraints on symbolic variables referred to in controllable actions, then the constraint conditions are evaluated and the resulting boolean value is passed to the special method Assume. This is how Pex is informed about the model's constraints over the data it just provided. If the constraint predicate evaluates to **false**, Pex will abandon this execution of the test code, and attempt using its constraint solver to compute data that does fulfill the constraint. This is an instance of a data choice, as explained in section 4.

Then, an observation representing the return from this call is recorded (line 53). The whole process is performed in the context of a transaction (lines 51-54), so that testing can be stopped if a conformance error is found. Otherwise, the current state is set to the transition's target (line 55).

Had the transition been labeled with a non-void method, the return value (with any output arguments) would have been stored in a local variable and added as part of the Return observation.

In the actual generated code, the method is invoked inside a **try**−**catch** block, so that a Throw observation can be added in case the call results in an exception. The exception object is stored as part of the observation, to be checked in constraints following a transition labeled with such an observation.

```
57 void StateS8()
58    ControllablesS8()
59
60 void ControllablesS8()
61    int choice = Manager.ChooseStep(2);
62    if (Manager.BoundReached)
63       finished = failed = true;
64    else
65       switch (choice)
66          case 0: StateS8Controllable0();
67          case 1: StateS8Controllable1();
68
69 void StateS8Controllable0()
70    Manager.BeginTransaction("InputTransaction(Cancel) → S10");
71    ATM.InputTransaction(Cancel);
72    Observation.AddReturn("InputTransaction");
73    if (Manager.EndTransaction())
74       currentState = "S10";
75    else finished = failed = true;
76
77 void StateS8Controllable1()
78          ...
```

State S8 (lines 57-78) is an example of a controllable action choice, as explained in section 4. The test manager is asked to select one out of two possible controllable steps (line 61) and, provided no exploration bound has been reached, the **switch** statement in line 65 routes the control flow to the method containing the chosen transition. The body of each of these methods (shown for the first of these steps in lines 69-75) is equivalent to the one already explained for state S0 in lines 47-56.

```
79 void StateS19()
80    finished = failed = !ObservablesS19();
81
82 void ObservablesS19()
83    return Observation.Observe(QuiescenceTimeout,
84       new Pattern(StateS19Pattern0, "TryWithdrawal"),
85       new Pattern(StateS19Pattern1, "TryWithdrawal")
86    );
87
88 bool StateS19Pattern0(Observation obs)
89    Manager.BeginTransaction("TryWithdrawal(1,10)/true → S20");
90    Manager.Assert((int)obs.Arguments[0] == 1);
91    Manager.Assert((int)obs.Arguments[1] == 10);
92    Manager.Assert((bool)obs.Arguments[2] == true);
93    if(withdrawal.HasValue)
94       Manager.Assert(withdrawal==10);
95    if(ok.HasValue)
96       Manager.Assert(ok.Value);
97    if(Manager.EndTransaction())
98       currentState = "S20";
99       return true;
100   else
101      return false;
```

```
102 bool StateS19Pattern1(Observation obs)
103    Manager.BeginTransaction("TryWithdrawal(1,10)/false → S21");
104    Manager.Assert((int)obs.Arguments[0] == 1);
105    Manager.Assert((int)obs.Arguments[1] == 10);
106    Manager.Assert((bool)obs.Arguments[2] == false);
107    ...
```

State S19 (lines 79-107) handles an observable action choice. It does so by creating two *observation patterns* in lines 84 and 85. The result of having observed this patterns (before the quiescence timeout described in section 4) is used in line 80 to decide whether a conformance error has been detected (and hence test-case execution must stop). Each pattern is constructed from a method (here, StateS19Pattern0 and StateS19Pattern1) that will be called by method Observation.Observe on actions observed from the SUT, in order to determine how to proceed.

Let us now analyze the body of method StateS19Pattern0, in lines 88-101, corresponding to transition TryWithdrawal(1,10)/**true** → S20, as seen in line 89. Arguments contained in the observation object are cast to the right types and compared to their expected values in lines 90-92. Should any of the transition arguments had been a variable, an assignment to the corresponding Variable object would have been generated, resulting in either a comparison to its previous value or the assignment proper (depending on whether the variable had been previously bound). In lines 93-96, target state constraints are verified, provided their variables are all bound. Finally, in lines 97-101, the transition to the target state is performed (in case all checks have succeeded) by setting the new current state and returning the result of the observation processing. The call to method Manager.EndTransaction() (line 97) rolls back all assignments if needed.

Method StateS19Pattern1, starting in line 102, only differs from State19Observation0 in that its expected return value is **false** instead of **true**, and its target state is S21 instead of S20 (not shown here).

5. RELATED WORK

Model coverage is a technique commonly used in test selection algorithms of model-based testing tools. Its goal is to generate a set of test cases large enough to cover every state/branch/path (and so on) in the model. This technique, which applies to finite state machine models as well as other kinds of models, is folklore, and we refer to [23] for a comprehensive text book overview. However, using *implementation coverage* to enhance tests derived from models has, to the best of our knowledge, never been tried before.

A technology like Pex and similar ones, such as those described in [9, 18], can in principle achieve results comparable to those of our approach. However, the "model" would need to be written as a program (or a unit test), using operational constructs (like, for example, loops over symbolic guards) in order to match the power of more declarative regular model notations. In contrast, our approach allows, via the framework provided by Spec Explorer 2007 [10] and action machines [13], to employ high-level modeling notations, such as DSL/UML. The same pitfalls appears in approaches like the one in [25]. Another advantage of our approach is that symbolic evaluation is applied only to individual paths (selected by model exploration) and not to the entire program, avoiding problems such as loops that often make the symbolic evaluation of a program infeasible.

In [2] we find a case study on the combination of model-based testing and runtime verification. Test inputs and a test oracle are generated by systematically exploring a model of the software's input domain, using a software model checker with extensions for symbolic execution. Then, code instrumentation techniques are used to inject the test oracle into the software under test. Unlike our approach, code coverage of the implementation is not taken into account during the process of test input generation. This prevents finding bugs, like the one in our running example, that only appear when certain implementation-dependent values are provided.

6. CONCLUSIONS

In this article, we report a novel testing approach we term AGBT. Preliminary evidence of feasibility, provided by a proof-of-concept implementation and sample, supports our conjecture that this approach has some advantages over traditional unit testing (either parameterized or not) and model-based testing:

1. In model-based testing, exploration is driven in essence by relevant traversals of the specification artifact, so any attempt at state-space coverage would cover model states rather than concrete states of a particular implementation. An alternative approach would be to take the actual code as the object of exploration, as is done in techniques such as software model checking. Nevertheless, the results of applying these techniques in practice show that (a) completely covering the state-space is not feasible, (b) effective state-space coverage criteria in this context are still subject of research (e.g. [14]), especially if one includes the choice of the tested call sequences, and (c) closing the environment, particularly in terms of data values, is a daunting task (see [21]). AGBT, instead, can be used to blend any reasonable model-based coverage criteria and/or test purposes together with traditional code coverage criteria. Besides, data values may be left symbolic, letting the engine provide relevant values. This may be the basis for defining interesting measures of thoroughness of testing and an effective bug finding technique as shown in our running example.

2. Model-based tools such as Spec Explorer, allow expressing end-to-end functional properties in a rather partial fashion. The combination of model-based symbolic exploration and parameterized unit testing results in a means to force the execution of code-coverage-maximizing, specification-relevant behavior by synthesizing the appropriate data values. Thus, we can think of this blend as both (a) a way to check the ability of a given unit under test to contribute to the achievement of a larger scope goal, and (b) a way to chose data values for model exploration (i.e., closing the model) that are also significant in terms of internal behavior of certain pieces of code.

3. The lack of oracles beyond assertions or absence of crashes limits the application scope of the ideas underlying unit testing (and parameterized) techniques. In fact, it is recognized that semantic-based faults are the most common kind of bugs (see [16]). Lightweight assertions encountered in practice tend to predicate on code-level abstractions and are likely to be too loose in terms of the actual role played by the neighboring piece of code in the larger context of the problem domain. Though code tricks and instrumentation to encode properties which are not naturally expressed as code assertions are possible, they are not a good engineering practice, as they are error prone and produce artifacts that are difficult to trust and to maintain (this is especially true in a setting where conformance goals should be clearly stated and enforced). On the other hand, model based is essentially about lightweightly and neatly defining and manipulating formal oracles, test purposes and scenario control specifications which may be, to a large extent, implementation independent. Thus, separation of concerns besides being a sensible and accepted principle, has a practical impact on the way models and code are developed and maintained.

4. Model-based notations, unlike traditional code assertions, deal with reactive specifications of behavior. Observing or forcing (through scenario-control constructs) certain reactions of the SUT is a key feature added to the unit test setting. More precisely, on the one hand, sophisticated set ups –beyond sequences of method invocations– may be defined in which the model operationally or declaratively dictates behavior that must be observed before the method under test is invoked (e.g., let's see how method m behaves after authentication has succeeded but a session timeout has occurred). Thus, in cases where brute force techniques are out of discussion, interesting scenarios involving relevant application mechanisms, patterns and/or environment behavior can be enforced (at different levels of abstraction) by model-based artifacts, in order to test the behavior of a method in context. On the other hand, model-based notations can capture the expected reactive behavior of a component. These assertions on expected interaction (e.g., correct use of design patterns, invariant-preservation protocol, etc.) lift parameterized unit testing techniques to the status of a parameterized protocol-conformance testing framework.

7. REFERENCES

[1] R. Alur, T. A. Henzinger, O. Kupferman, and M. Vardi. Alternating refinement relations. In *Proceedings of the 9th International Conference on Concurrency Theory (CONCUR'98)*, volume 1466 of *LNCS*, pages 163–178, 1998.

[2] C. Artho, H. Barringer, A. Goldberg, K. Havelund, S. Khurshid, M. Lowry, C. Pasareanu, G. Rosu, K. Sen, W. Visser, and R. Washington. Combining test case generation and runtime verification. *Theor. Comput. Sci.*, 336(2-3):209–234, 2005.

[3] M. Barnett, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes. Towards a tool environment for model-based testing with AsmL. In Petrenko and Ulrich, editors, *Formal Approaches to Software Testing, FATES 2003*, volume 2931 of *LNCS*, pages 264–280. Springer, 2003.

[4] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2001.

[5] K. Beck. *Test-Driven Development: By Example*. Addison-Wesley, 2002.

[6] C. Campbell, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes. Testing concurrent object-oriented systems with spec explorer. In J. Fitzgerald, I. J. Hayes, and A. Tarlecki, editors, *FM 2005: Formal Methods, International Symposium of Formal Methods Europe*, volume 3582 of *Lecture Notes in Computer Science*, pages 542–547. Springer, 2005.

[7] L. de Alfaro and T. A. Henzinger. Interface automata. In *Proceedings of the 8th European Software Engineering Conference and the 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 109–120. ACM, 2001.

[8] E. Gamma and K. Beck. JUnit: A regression testing framework, 2001. http://www.junit.org.

[9] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *ACM SIGPLAN 2005*

Conference on Programming Language Design and Implementation (PLDI'05), pages 213–223, 2005.

[10] W. Grieskamp. Multi-paradigmatic model-based testing. In K. Havelund, M. Núñez, G. Rosu, and B. Wolff, editors, *FATES/RV 2006: Formal Approaches to Testing and Runtime Verification*, volume 4262 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2006. invited contribution.

[11] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from abstract state machines. In *ISSTA'02*, volume 27 of *Software Engineering Notes*, pages 112–122. ACM, 2002.

[12] W. Grieskamp and N. Kicillof. A schema language for coordinating construction and composition of partial behavior descriptions. In J. Whittle, L. Geiger, and M. Meisinger, editors, *SCESM*, pages 59–66. ACM, 2006.

[13] W. Grieskamp, N. Kicillof, and N. Tillmann. Action machines: a framework for encoding and composing partial behaviors. *International Journal of Software Engineering and Knowledge Engineering*, 16(5):705–726, 2006.

[14] R. Grosu and S. A. Smolka. Monte carlo model checking. In N. Halbwachs and L. D. Zuck, editors, *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 271–286. Springer, 2005.

[15] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[16] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have things changed now? – An empirical study of bug characteristics in modern open source software. In *ICSE: Proceedings 29th International Conference on Software Engineering*, 2007.

[17] H. Robinson. Finite state model-based testing on a shoestring. In *Proceedings of the International Conference on Software Testing Analysis and Review (STARWEST 1999), Software Quality Engineering, San Jose, CA, USA, October 1999*.

[18] K. Sen and G. Agha. Cute and jCUTE : Concolic unit testing and explicit path model-checking tools. In *18th International Conference on Computer Aided Verification (CAV'06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 419–423. Springer, 2006. (Tool Paper).

[19] K. Stobie. Model based testing in practice at microsoft. In *Proceedings of the Workshop on Model Based Testing (MBT 2004)*, volume 111 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2004.

[20] N. Tillmann and W. Schulte. Unit tests reloaded: Parameterized unit testing with symbolic execution. *IEEE software*, 23:38–47, 2006.

[21] O. Tkachuk, M. B. Dwyer, and C. S. Pasareanu. Automated environment generation for software model checking. In *ASE*, pages 116–129. IEEE Computer Society, 2003.

[22] J. Tretmans and A. Belinfante. Automatic testing with formal methods. In *EuroSTAR'99: 7th European Int. Conference on Software Testing, Analysis & Review*, Barcelona, Spain, November 8–12, 1999. EuroStar Conferences, Galway, Ireland.

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

[24] G. Venolia, R. DeLine, and T. LaToza. Software development at microsoft observed. Technical Report MSR-TR-2005-140, Microsoft Research, October 2005.

[25] W. Visser, C. S. Pasareanu, and R. Pelánek. Test input generation for red-black trees using abstraction. In D. F. Redmiles, T. Ellman, and A. Zisman, editors, *ASE*, pages 414–417. ACM, 2005.