Automatic Generation of Test Models and Properties from UML Models with OCL Constraints

Miguel A. Francisco
Interoud Innovation S.L. (Spain)
miguel.francisco@interoud.com

Laura M. Castro
MADS Group – Dept. Computer Science
University of A Coruña (Spain)
lcastro@udc.es

ABSTRACT

Model-Based Testing and Property-Based Testing are two testing methodologies that usually facilitate the automation of the generation of test cases, using either models or properties as basis to derive complete test suites. In doing so, they also contribute to reduce both the source code needed for testing purposes and the time spent on writing those tests, hence saving effort and increasing maintainability.

In this paper, we describe how to generate complete property-based test suites automatically translated from a model that describes the system under test, specifically a UML model with OCL constraints. The proposed approach can be applied to both stateless and stateful components.

We have used QuickCheck, an automatic property-based testing tool for test case generation, execution and diagnosis, as support tool. Our method produces QuickCheck executable test models as output, so we use the same tool to run and evaluate them. As part of our work, we analyze and discuss the advantages and disadvantages of our approach in contrast to writing test suites manually.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

reliability, verification, testing

Keywords

UML, OCL, QuickCheck, black-box testing, functional testing, model-based testing, property-based testing

1. INTRODUCTION

Testing is a key activity in software development, meant to verify the behavior of a program. One way of doing so is executing a finite set of test cases in order to detect failures, and comparing the result of the real execution with some expected result or specification. There are many approaches to testing, some of them only adapted for certain kind of requirements (functionality, usability, robustness, performance, integration, etc.).

Among the existing testing methodologies, Property-Based Testing (PBT [15]) is based on the use of properties to specify the constraints to be fulfilled by the system under test (SUT). Such properties are often used to produce individual test cases, a task which, when automated, represents a great improvement over other approaches where test cases have to be manually specified one by one. However, this usually comes at the price of describing the SUT specification as properties. We can alleviate this by combining PBT with Model-Based Testing (MBT [28]), in which test cases are generated from a black-box model that describes some (usually functional) aspects of the SUT. Thus, we could generate properties and test cases from such MBT models.

This work describes how we have combined PBT and MBT for improving functional testing in terms of effort. The result of combining MBT and PBT is a new testing approach in which testers get the benefits of using properties to generate test cases, and those of specifying the test requirements using a model of the SUT. In particular, we have used UML models complemented with OCL constraints as starting point, so that testers do not have to deal either with the implementation details of the SUT, nor the peculiarities of a new formalism to specify the tests.

The rest of the paper is organized as follows. Section 2 describes the state of the art, analyzing different approaches which could be used for the same purpose. Section 3 explains the architecture of our approach, describing the components involved. Sections 4 and 5 describe how to apply the approach to test stateless and stateful components, respectively. Section 6 discusses the results and limitations of the approach. Finally, in Section 7 conclusions and future lines related to this work are included.

2. STATE OF THE ART

The main goal of research in testing automation is to reduce the effort that testing tasks require, and hence to provide more systematic, effective testing methods while increasing its effectivity. Testing approaches can be classified into five different automation levels [21].

In the first level of such automation scale we find Testing-by-Contract. Testing-by-Contract [10,21] is a testing approach that uses assertions inlined with the source code to be able to evaluate test cases. These assertions were introduced by Bertrand Meyer as part of a development approach called Design-by-Contract [22,23], in which the main idea is to use a contract to describe the responsibilities of the classes in an object-oriented design using preconditions, postconditions and class invariants. Apart from the Eiffel programming language, which was the pioneer in implementing Design-by-Contract, there are a few utilities that provide support for it to other programming languages, such as JML [20], iContract [19], jContractor [18], or jContract [25] for Java, or Spec# [5] for C#. Some of these utilities also provide tools to use the defined preconditions and postconditions to test operations with lots of randomly generated input values, such as JMLUnit [9] (JML) or JTest [25] (jContract).

These tools allow one to test a component following a MBT approach [28,29]. Rather than manually writing tests based on the requirements documentation, MBT approaches use a model that captures (some of) the requirements of the SUT and defines the expected SUT behavior. The model is a schematic and/or partial description of a system, and it can be used to generate tests automatically. Thus, it is important to decide a good level of abstraction for the model, including the aspects that are important for test generation.

There are many notations to define models [29], such as state-based notations and transition-based notations, and there are several formalisms that may be used [16], UML being one of them [31]. What is more, UML Testing Profile [4], a UML profile for testing purposes, was added to the UML standard in order to facilitate the use of UML for specifying testing requirements. UML [26] is usually combined with OCL [30] to specify preconditions and postconditions. These constraints can be used to generate executable code [7,24] that tests if the SUT satisfies those conditions using automatically generated input test data [1,6].

On the other hand, PBT [15] approaches implement algorithms to generate test cases from properties [11]. Properties are declarative statements that describe SUT requirements, and are often written in specific-purpose languages. As statements, properties are generally closer to the abstraction level of specifications, and thus are easier to produce than models. Our proposal consists in using a standard UML+OCL model to specify the SUT requirements to test, and generate properties from that model, using then a PBT tool to generate random input data for those properties and to check if the SUT satisfies those requirements.

3. PROPOSED TESTING ARCHITECTURE

In MBT, a model of the SUT is built from the SUT requirements. This model, together with a test selection criteria that says which kind of tests must be generated, is used by a test suite generator to generate an *Abstract Test Suite* (ATS) with abstract test cases that must be *instantiated*. There are several possible approaches to derive specific test cases from abstract test cases [28], such as implementing an adapter, transforming the ATS into executable test scripts or using a mixed approach that combines both. Finally, tests are executed and test results are analyzed.

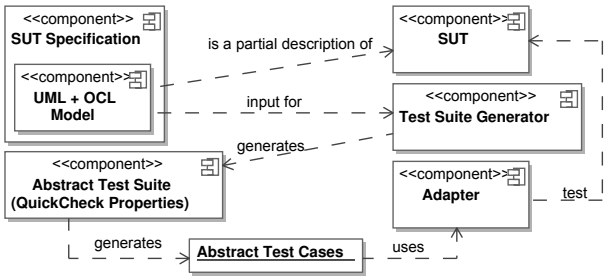

Figure 1: Proposed Testing Architecture

Our approach respects this workflow (cf. Fig. 1). We have chosen UML [26] to write the model that describes the SUT requirements. Of course, the better the model, the higher the quality of the generated tests. Hence, it is very important that the model is as detailed as possible, but also that it ignores details that are not test-wise important. The choice of UML was motivated by it being a standardized general-purpose modeling language, widespread in industry. In contrast to other alternatives such as JML or Spec#, UML is implementation-independent. To complement the UML model with the constraints that it cannot capture, we chose the declarative language OCL [30]. We use OCL to describe rules in different UML diagrams (class diagrams, sequence diagrams, state machine diagrams, etc.), in particular preconditions and postconditions for methods/operations, and class/component invariants.

Also, our ATS is composed of abstract properties automatically generated from the model by our *Test Suite Generator*. To do that, this component analyzes the input model, using Dresden OCL [13, 14] to parse the UML model and OCL constraints, and generates an ATS from the specification described by the model. The abstract specification is a set of declarative properties written for QuickCheck [3, 12], a versatile testing tool that can generate random test cases from a set of such properties and a series of data generators defined by the tester. Depending on the model, the *Test Suite Generator* produces the most suitable ATS: for stateless components, universally quantified properties are generated; for stateful components, they are written as a state machine.

Thus, our approach is based on the use of a standard language, UML, to avoid testers to deal with the details of specific implementation languages and focus on modeling test requirements. The ATS that is generated from the model does not depend on the SUT implementation either, so it is reusable for testing different SUT implementations. To derive specific test cases from the ATS and run them against a given SUT, we use a generic adapter (cf. Fig. 1). For example, for SUTs written in Java, we use Jinterface [17].

4. STATELESS COMPONENTS

Stateless components do not have an internal state that is modified by the execution of the functionalities they offer. Thus, functions to test in stateless components do not have side effects, and are independent of each other. The result of invoking those functions does not depend on when they are called, but only on the input parameters. Libraries are usually good examples of stateless components.

In our approach, OCL constraints are used along with a UML model to generate an ATS composed by QuickCheck properties. The UML model is used to know the operations to be tested for a given component, as well as the details of the data types of the parameters and return values of each function. OCL constraints are used to generate test oracles, that is, the expected output values that correspond to accepted input values.

To test a function f that receives N parameters $p_1, \ldots, p_N$, we generate a property

$$\forall\, \overline{p} \in G,\ pre(\overline{p}) \rightarrow\ r = f(\overline{p}) \wedge post(\overline{p}, r)$$

where $\overline{p}$ are the input parameters $p_1, \ldots, p_N$ of f, and N is the number of parameters; G are data generators $G_1, \ldots, G_N$ for $p_1, \ldots, p_N$; *pre* is the set of constraints on the input parameters (i.e. preconditions); and *post* is the set of constraints on the return value (i.e. postconditions, body constraint, or invariant). That property is declaratively written in QuickCheck as:

```
?FORALL(P, ?SUCHTHAT(P, G(), pre(P)), begin
    R = f(P), post(P, R)
  end).
```

For example, if we have a `ListUtils` library and a `max` operation with the following OCL specification:

```
context ListUtils::max(l: Sequence(Integer)): Integer
pre not_empty: l -> notEmpty()
post max: l -> forAll(x : Integer | result >= x)
```

the following QuickCheck property is generated:

```
prop_listutils_max()->
  ?FORALL(L, ?SUCHTHAT(L, ocl_gen:gen_sequence_integer(),
        ocl_seq:not_empty(L)),
    begin
      Result = listUtils:max(L),
      ocl_seq:forAll(fun(X) -> (Result >= X) end, L)
    end).
```

QuickCheck can run this `prop_listutils_max` property, generating random lists of integers using the `ocl_gen:gen_sequence_integer` data generator, discarding those lists that do not satisfy the `ocl_seq:not_empty` precondition. The lists that satisfy the precondition, are then used as parameters to the `listUtils:max` SUT function. Finally, the actual result of the real implementation of the tested operation is checked for the `Result >= X` postcondition.

In general, the testing process of a stateless component is:

1. A component from the input UML model is selected.
2. An operation from the selected component is chosen. Since the component is stateless, the order in which the operations are chosen is not important.
3. Specific input parameter values are generated.
4. Preconditions (if they exist, written in OCL) are used to validate the generated input parameters. If they hold, the operation is executed and result from real SUT is obtained. Otherwise, we go back to step 3.
5. Postconditions, body conditions, or invariants (if they exist, written in OCL) are checked. If they hold, we go back to step 2. Otherwise, the testing process is interrupted: an error has been found.
6. The process ends when all the operations of all the components have been tested.

QuickCheck properties represent abstract test cases from which specific test cases must be produced. In our case, this process is performed by a combination of the QuickCheck tool execution itself, according to the previous workflow, and an adapter that invokes the real SUT transforming the specific values that QuickCheck generates using the data generators into a data format that the real SUT understands, also transforming the SUT return values into QuickCheck format again for test evaluation.

Thus, QuickCheck not only automatically generates a large amount of specific test cases with randomized input values from the properties. QuickCheck runs those test cases against a given real SUT, and diagnoses its behaviour in a black-box manner. Furthermore, whenever an error is found, QuickCheck *shrinks* the input values to find a smaller specific test case that causes the same error [2, 8, 32]. Hence, our approach replaces the tedious task of manually designing large test suites by the definition of a UML+OCL model, and enables automatic generation and execution of tests, together with a significant improvement of debugging thanks to small counterexamples.

5. STATEFUL COMPONENTS

Stateful components have an internal state which may be modified by the execution of the functionalities they offer. In addition, the actual behaviour of their operations may also depend on their internal state. Therefore, the state of a stateful component must be taken into account for testing purposes, i.e. preconditions and postconditions depend on the component state, and each test case must include sequences of different operations, taking into account the order in which they are executed to diagnose the final result.

Taking this into account, the steps to follow are quite similar to those showed for stateless components. In addition, we need to save some information after operation execution (and corresponding postcondition evaluation) and make it available to the whole test case, should we need it to check the preconditions of subsequent operations. To support this kind of stateful testing, QuickCheck provides a mechanism to group declarative properties for the operations of a stateful component in the shape of a state machine definition.

A QuickCheck state machine has as transitions the set of operations of the stateful component to be tested (again, from a black-box point of view). Each operation can have preconditions and postconditions. Besides, we indicate which data is to be stored as internal state, and specify both the initial value and how the contents of the internal state are modified by the execution of each operation. Once we have written the properties for the stateful component in this structured way, QuickCheck can run them, generate multiple test sequences, execute them against a real stateful SUT and evaluate whether it conforms to the corresponding specification.

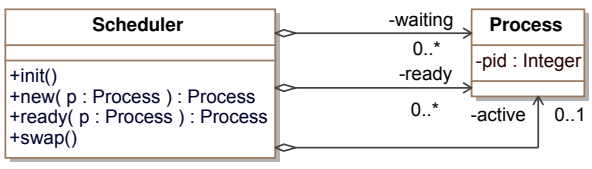

Figure 2: Scheduler UML class diagram

Fig. 2 shows an example of a stateful component, specifically a process scheduler [6,27]. The static UML information can be complemented by an OCL specification (we deal with the `swap` operation in Sect. 5.1):

```
context Scheduler
inv: (ready->intersection(waiting))->isEmpty()
    and not ((ready->union(waiting))->includes(active))
    and (active = null implies ready->isEmpty())

context Scheduler::init()
post: (ready->union(waiting)) = Set{} and active = null

context Scheduler::new(p:Process): Process
pre: p <> active
    and not (ready->union(waiting))->includes(p)
post: waiting = (waiting@pre->including(p))
     and ready = ready@pre and active = active@pre
     and result = p

context Scheduler::ready(p:Process): Process
pre: waiting->includes(p)
post: waiting = waiting@pre->excluding(p)
     and if active@pre = null then
              (ready = ready@pre and active = p)
         else (ready = ready@pre->including(p)
               and active = active@pre)
         endif and result = p
```

With this information (the UML class diagram and the OCL specification), the *Test Suite Generator* must generate the QuickCheck state machine.

The first step is to identify the operations to test. This information is obtained from the UML class diagram, in which the signature of each operation (name, input parameters and return value) is specified. After that, preconditions and postconditions for each operation have to be defined. This information is available in the OCL specification. In addition, we also need to determine which data is to be stored in the internal state, and how it will be updated after the execution of each operation. For the schedule example:

- The commands (i.e. transitions) are `init`, `new` and `ready`. They are listed in a symbolic way (i.e. `{call, {COMPONENT, OPERATION, ARGUMENTS}`) to be able to manipulate them prior to actual execution:

  ```
  command(S) -> oneof([
    {call, scheduler, init,  []},
    {call, scheduler, new,   [gen_process()]},
    {call, scheduler, ready, [gen_process()]}]).
  ```

  Same as for testing of stateless components, the input parameters that the operations receive are generated using data generators. In this case, generator `gen_process` produces a new *abstract process instance*. The abstract representation chosen for processes at the testing level is a tuple with the attributes and their corresponding values. For example, if we have a component C with n attributes $a_1, \ldots, a_n$ with values $v_1, \ldots, v_n$, the abstract representation of an instance of such component is the tuple: `{c, [{a1, v1},...,{an, vn}]}`. Thus, according to Fig. 2, the data generator for processes is generated as follows:

  ```
  gen_process() -> {process, [{id, eqc_gen:int()}]}.
  ```

- The internal state (`ts`) stores information that appears at some postcondition with the `@pre` operation, which in OCL references a value before operation execution:

  ```
  -record(ts, {active,ready,waiting}).
  ```

  The initial value for the internal state must also be set. However, this information is not present in the OCL specification, so it will be obtained from the SUT (which must be initialized before starting the test):

  ```
  initial_state()-> #ts {
    active = scheduler:get_active(),
    ready = scheduler:get_ready(),
    waiting = scheduler:get_waiting()}.
  ```

- Each precondition is generated from the corresponding OCL precondition. For example, for the `new` operation:

  ```
  precondition(S, {call, scheduler, new, [P]})->
    (ocl:neq(P, S#ts.active)
      andalso not(ocl_set:includes(P,
        ocl_set:union(S#ts.waiting, S#ts.ready))));
  ```

- Each postcondition is generated from the corresponding OCL postcondition and global invariant (if any). For example, for the scheduler `new` operation:

  ```
  postcondition(Pre, After,
                {call, scheduler, new, [P]}, R)->
    ocl_set:eq(After#ts.waiting,
        ocl_set:including(P, Pre#ts.waiting)) andalso
     ocl_set:eq(After#ts.ready, Pre#ts.ready) andalso
     ocl:eq(After#ts.active, Pre#ts.active) andalso
     ocl:eq(R, P) andalso invariant(Pre, After);
  ```

- The information to generate the functions to update the internal state can be extracted from the OCL postconditions. For example, the postcondition for the `new` operation says that after executing this operation the `waiting` processes will be the previous `waiting` processes including the new created process `p`; also, the `ready` and `active` processes will not change:

  ```
  next_state(S, R, {call, scheduler, new, [P]})->
    S#ts { active = S#ts.active, ready = S#ts.ready,
      waiting = ocl_set:including(P, S#ts.waiting)};
  ```

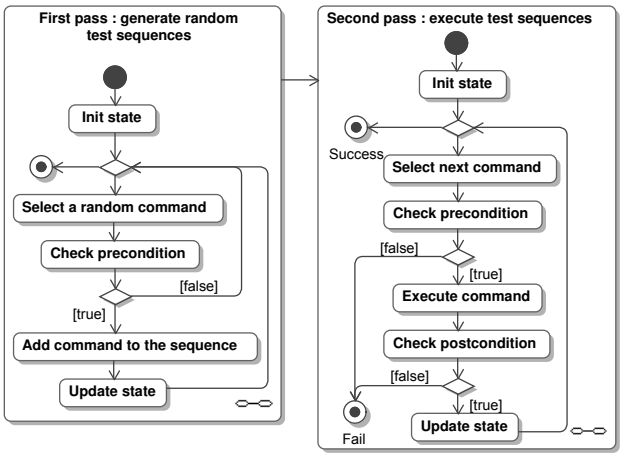

Figure 3: QuickCheck state machine test execution

Once the properties for the stateful component have been generated from the UML+OCL model like this, QuickCheck

can generate specific test cases from them, random sequences of commands with specific parameter values that satisfy the preconditions (cf. Fig. 3). The test sequence is generated working with symbolic values and checking only the preconditions. Only once the whole sequence has been generated it is actually executed. During that second pass, when the internal state is updated, it may be the case that the precondition or postcondition for some operation does not hold, which would mean we found a violation of the SUT specification. Fig. 4 shows an example of the abstract test sequence that QuickCheck could generate for the scheduler example.

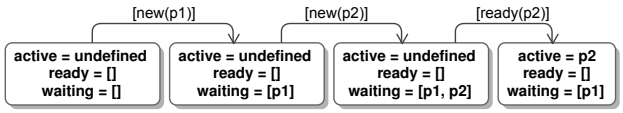

Figure 4: Example of stateful test sequence

Same as for stateless components, we also need to translate the QuickCheck data format to a format the real SUT understands in order to successfully call the actual implementation. For this purpose, we use a reusable adapter dependent on the implementation language of the SUT (cf. Fig. 1).

5.1 Partially-Specified Components

In some situations, we may have to deal with partial or incomplete specifications. For example, let us consider the `swap` operation, with the following specification:

```
context Scheduler::swap()
pre: active <> null
post: if ready@pre->isEmpty() then
        (active = null and ready = Set{})
      else (ready@pre->includes(active) and
            ready = ready@pre->excluding(active))
      endif and
      waiting = waiting@pre->including(active@pre)
```

According to this valid OCL definition, the `swap` operation selects a process from the list of `ready` processes (provided it is not empty), which will be the next `active` process (`ready@pre->includes(active)`), and adds the previous `active` process to the list of `waiting` processes. However, *nothing is said as to which of the* `ready` *processes is to be selected*. In other words, the scheduling criteria is not fully specified, which makes it impossible to generate the `next_state` function for the `swap` operation.

In our approach, we have implemented the *Test Suite Generator* with the ability to detect these situations. When it does, it follows a different approach to generate the stateful properties: instead of using the two-pass algorithm which symbolically analyzes the preconditions in the first pass, it does not define any preconditions and consequently skips directly to the second pass: `precondition(_S, _C)-> true.`

In the second pass, we use a QuickCheck-provided mechanism to define *dynamic preconditions*, that is, preconditions that are temptatively executed before deciding if certain operation is eligible to be executed. In other words, the two passes are merged into only one, where the sequence is built *online*: the sequence of operations is chosen as the text sequence execution progresses. Dynamic preconditions are defined in a similar way to regular ones, as seen in Sect. 5:

```
dynamic_precondition(S,{call,scheduler,swap,[]})->
  ocl:neq(S#ts.active, undefined);
```

The internal state update at the `next_state` function needs in these cases (where the behaviour is not specified) to use the result of the real SUT execution (`R`). After executing the `swap` operation, the next `active` process will be obtained from the SUT, and the state will be updated accordingly. This is not a ideal situation, since we would rather be able to determine the next active process at the model level and not trust the SUT, but it enables us to perform automatic PBT even from an incomplete model.

6. DISCUSSION

We have described the two scenarios we may face when attempting to generate testing properties from a UML+OCL specification: stateless (cf. Sect. 4) and stateful (cf. Sect. 5) components. We have also presented how to generate testing properties for stateful components in two situations: when we have a complete specification (i.e. offline generation of test cases from a QuickCheck state machine), and when we do not (i.e. online generation of test cases from a QuickCheck state machine with dynamic preconditions).

For the sake of generality, one could argue that the online strategy that uses the dynamic preconditions instead of the regular preconditions of a QuickCheck state machine can actually be used in all cases. However, there is a strong reason for using regular preconditions and the two-pass algorithm: *shrinking*. QuickCheck's shrinking capabilities are of great help when an error is found, because they provide smaller counterexamples that, causing the same error as the original failing test case, make error debugging and fixing easier [2, 8, 32]. However, QuickCheck's shrinking capabilities are dependent on the offline generation of test cases, since the shrinking algorithm manages the test cases in their symbolic format, which is produced in the first pass (cf. Fig. 3). Thus, we should only use online generation when it is required by the presence of an incomplete specification.

The current version of our *Test Suite Generator* generates code for data generator functions that build data with the required data type. For example, the generator for processes generates processes with random identifiers. This could be improved, because there are cases in which pure randomness may not be useful. For example, the precondition of the `ready` operation indicates that it can be only executed if it receives a `waiting` process. Since the `gen_process` function produces only processes with random IDs, the `ready` operation will be executed very few times, only when the generated random process happens to be a `waiting` process. An improvement would be to take preconditions into account in the generation process, so that more suitable data can be built. That is, for OCL preconditions that use the `includes` operator, the data generator could balance the creation of data included in the specified term, as well as data which is not included (thus combining positive and negative testing).

Finally, our *Test Suite Generator* does not perform any optimization in the generation of the QuickCheck state machine at the moment. However, the inclusion of a pre-processing stage would allow us to make this component smarter by performing certain analysis of the given specification before

generating the properties. For example, the postcondition for the `init` operation checks that the `union` of the `ready` and `waiting` processes is the empty set, which implies that both `ready` and `waiting` will be empty sets after executing the operation. Should our *Test Suite Generator* be able to detect this, it could formulate the postcondition in a different way, and even avoid the use of online test sequence generation in favour of the preferable offline generation.

7. CONCLUSIONS

This paper describes a MBT approach that takes advantage of a PBT tool. We transform a UML model with OCL constraints into properties to be used as an implementation-independent test suite, and then use QuickCheck to generate test cases from those properties, run them via a suitable adaptor for the technology of the SUT, and evaluate them.

The testing properties we generate are both an executable, up-to-date specification of the SUT, and are formulated as declarative statements for stateless components and state machines for stateful components. We have explained the limitations we found in the generation process of the testing properties, and also how we decided to overcome them.

The current version of the *Test Suite Generator* does not support all the OCL features defined in the OCL specification (currently 2.3.1), such as some built-in operations and OCL messages. As a future work, we plan to extend it to include more OCL features. Also, we want to improve data generators and include a pre-processing of the specification for possible optimizations. Finally, our future plan is to extend this approach to other kinds of testing, such as integration testing.

8. REFERENCES

[1] S. Ali, M. Z. Iqbal, A. Arcuri, and L. Briand. A search-based ocl constraint solver for model-based test data generation. In *11th International Conference on Quality Software*, QSIC '11, pages 41–50. IEEE Computer Society, 2011.

[2] T. Arts, L. M. Castro, and J. Hughes. Testing erlang data types with quviq quickcheck. In *ACM SIGPLAN Workshop on Erlang*, 2008.

[3] T. Arts, J. Hughes, J. Johansson, and U. Wiger. Testing telecoms software with quviq quickcheck. In *ACM SIGPLAN workshop on Erlang (ERLANG'06)*, New York, NY, USA, 2006. ACM Press.

[4] P. Baker, Z. R. Dai, J. Grabowski, O. Haugen, I. Schieferdecker, and C. Williams. *Model-Driven Testing: Using the UML Testing Profile*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[5] M. Barnett, K. Leino, and W. Schulte. The spec# programming system: An overview. pages 49–69. Springer, 2004.

[6] M. Benattou, J. M. Bruel, and N. Hameurlain. Generating test data from ocl specification. In *ECOOP Workshop on Integration and Transformation of UML models*, 2002.

[7] A. D. Brucker, M. P. Krieger, D. Longuet, and B. Wolff. A specification-based test case generation method for uml/ocl. In *International Conference on Models in Software Engineering*, MODELS'10, pages 334–348, Berlin, Heidelberg, 2011. Springer-Verlag.

[8] L. M. Castro and T. Arts. Testing data consistency of data-intensive applications using quickcheck. *Electronic Notes in Theoretical Computer Science*, 271:41–62, 2011. Proceedings of the 10th Spanish Conference on Programming and Languages (PROLE 2010).

[9] Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The jml and junit way. In *16th European Conference on Object-Oriented Programming*, ECOOP'02, pages 231–255, London, UK, 2002. Springer-Verlag.

[10] I. Ciupa and A. Leitner. Automatic testing based on design by contract. In *6th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World (Net.ObjectDays 2005)*, pages 545–557, 2005.

[11] K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *5th ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM.

[12] K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *International Conference on Functional Programming*, pages 268–279, 2000.

[13] B. Demuth. The dresden ocl toolkit and its role in information systems development. In *13th International Conference on Information Systems Development: Methods and Tools, Theory and Practice Conference, Advances in Theory, Practice and Education*, 9-11 September, 2004.

[14] B. Demuth, S. Loecher, and S. Zschaler. Structure of the dresden ocl toolkit extended abstract. In *2nd International Fujaba Days "MDA with UML and Rule-based Object Manipulation"*, September 15 - 17, 2004.

[15] J. Derrick, N. Walkinshaw, T. Arts, C. B. Earle, F. Cesarini, L.-A. Fredlund, V. Gulías, J. Hughes, and S. Thompson. Property-based testing - the protest project. *Lecture Notes in Computer Science*, 6286 LNCS:250–271, 2010.

[16] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos. A survey on model-based testing approaches: a systematic review. In *1st ACM international Workshop on Empirical assessment of Software Engineering Languages and Technologies (WEASELTech '07)*, pages 31–36, New York, NY, USA, 2007. ACM.

[17] Jinterface. `http://www.erlang.org/doc/apps/jinterface/index.html`, 2012.

[18] M. Karaorman, U. Hölzle, and J. L. Bruno. jcontractor: A reflective java library to support design by contract. In *2nd International Conference on Meta-Level Architectures and Reflection*, Reflection '99, pages 175–196, London, UK, UK, 1999. Springer-Verlag.

[19] R. Kramer. icontract - the java(tm) design by contract(tm) tool. In *Technology of Object-Oriented Languages and Systems (TOOLS '98)*, pages 295–307, 1998.

[20] G. T. Leavens and Y. Cheon. Design by contract with jml. www.jmlspecs.org, 2006.

[21] P. Madsen. Testing by contract - combining unit testing and design by contract. *Proceedings of the Nordic workshop on Software Development Tools and Techniques*, 2002.

[22] B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, Oct. 1992.

[23] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., second edition, 1997.

[24] R. Moiseev, S. Hayashi, and M. Saeki. Generating assertion code from ocl: A transformational approach based on similarities of implementation languages. In *12th International Conference on Model Driven Engineering Languages and Systems*, MODELS '09, pages 650–664, Berlin, Heidelberg, 2009. Springer-Verlag.

[25] Parasoft corporation. `http://www.parasoft.com/`, 2012.

[26] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[27] P. A. Salas and B. K. Aichernig. Automatic Test Case Generation for OCL: a Mutation Approach. Technical report, 2005.

[28] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

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

[30] J. Warmer and A. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., 2 edition, 2003.

[31] Y. Wu, M. Chen, and J. Offutt. Uml-based integration testing for component-based software. In *International Conference on COTS-Based Software Systems*, pages 251–260, 2003.

[32] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28:2002, 2002.