Model Based Functional Testing using Pattern Directed Filmstrips

Tony Clark
Centre for Model Driven Software Engineering
School of Computing
Thames Valley University
St Mary's Road, Ealing, London, UK, W5 5RF
tony.clark@tvu.ac.uk

Abstract

Model driven functional system testing generates test scenarios from behavioural and structural models. In order to autmatically generate tests, conditions such as invariants and pre-/post-conditions must be precisely defined. UML provides the Object Constraint Language (OCL) for this purpose; however OCL expressions can become very complex. This paper describes an approach that allows many commonly found OCL patterns to be expressed as snapshot patterns *that correspond directly to the information model diagrams. Behaviour is constructed as chains of snapshots, or* filmstrips*. Snapshots and filmstrips are as expressive as UML behaviour models and OCL but it is argued that they are more accessible and more modular.*

1 Introduction

It is important to integrate model based testing with development processes and to reuse models from the design processes where possible [9]. The facilities provided by UML are ideally placed to capture functional requirements. However, as described in [2], although models can be used for an explicit description of a test case, the functional requirements expressed in UML using representations such as use-case models and activity diagrams are often informal, have a weak semantics and are weakly integrated.

Design models are often detailed and can rapidly become out of date with respect to the implementation of a system. Tests generated from low level models can work directly on the implementation (such as JUnit in [3]), however the tests must be expressed in terms of implementation detail. The essential idea of model based testing is to compare an abstract specification to a concrete implementation [14]. Tests generated from models that describe high-level functional requirements and associated information structures [11] change much more slowly than design models.

Current approaches to model based testing do not integrate the process of test generation with the requirement models including use-cases and class models. As pointed out in [13], the process of deriving tests from requirements tends to be unstructured; artifacts that explicitly encode the intended behaviour can help minitgate the implications of these problems.

This paper proposes a novel approach and language for model based test generation that is based on integrating functional requirements models using snapshots and filmstrips. Section 2 describes the approach and motivates the technology; section 3 describes snapshot patterns as the basis of invariants and filmstrip steps; section 4 describes a language for filmstrips; finally 5 places the work in context and describes future directions.

2 Functional Testing and Filmstrips

Functional requirements for a system are often represented using UML use-case models that show the key functionality from an external viewpoint. The use case functionality consists of two key elements: the logical interface and the physical interface. The physical interface is concerned with the features of user iteraction with the system, for example GUI widgets. The logical interface is concerned with the events and data flow that occur between users and the system. Testing physical interfaces involves issues such as human factors which we do not address here; this paper is about testing logical interface functionality.

Use cases can be decomposed using «`extends`» and «`includes`» relationships. Ultimately, this decomposition can lead to a fine grain desription of the system functionality in terms of the events that flow between a user and the system. Use cases can be associated with use-case Scenarios that describe the information that flows between the use and the system. Scenarios are expressed as structured natural language and therefore cannot be processed by machine.

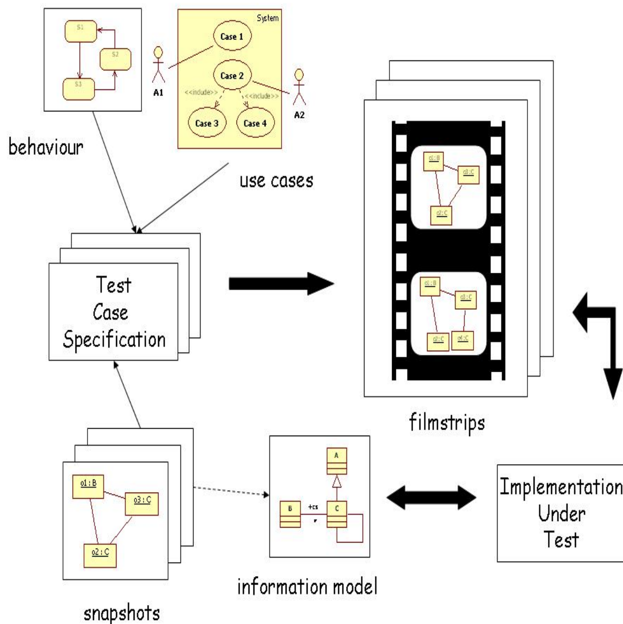

Figure 1. Testing Architecture

The information content of a system is often expressed as UML class models. The semantics of an information model can be (partially) expressed as a collection on invariants on classes. The invariants can be expressed as OCL expressions. Each class has a collection of operations. Each operation can be specified in terms of its effect, when called, on the information model. The effect can be captured precisely using pre- and post- conditions expressed using OCL.

Given an implementation of a system, it is desirable to generate tests in terms of the model that was used to specify and design it. Given a relationship between the information model (logical system view) and the implementation (physical system view) then the information model can be used to generate individual operation tests in terms of the correct changes to the information states expressed as pairs of *snapshots*. Furthermore, given behavioural models, such as state machines, it is possible to construct tests in terms of sequences of operations and the required information states. Sequences of snapshots produced by operations in this way will be referred to as *filmstrips*.

If the use case models are decomposed to a sufficiently fine grain level then there can be a one-to-one correspondence between the class operations in the information model and the use cases. This is attractive because it provides a seamless development process from initial requirements through to tests that can be applied to an implementation. Figure 1 shows the key features of the approach. Tests described in this way are shielded from design and implementation decisions via a mapping. Changes to the implementation will often require only minor changes to the mapping and therefore leave the functional tests intact.

Consider the requirement on modelling technologies that would support figure 1. We will limit ourselves to UML. If use cases are used to represent logical information operations then they do not provide any mechanism for precisely defining arguments and control flow. This information can be expressed in terms of other UML models, however the information in use-case models cannot be linked to other models in any precise way. It would be useful to minimlly connect use-cases to class operations to support an integrated development process.

Snapshots are defined as collections of objects that satisfy some conditions. Object models can be used to describe configurations of objects, but they are *ground* in the sense that there is no semantics for object diagrams with variables. Generality can be accommodated in terms of class models, however structural constraints have to be expressed in general terms using OCL which is either *too* general (applies to all) or is very complex when the same information could be made more accessible using object diagrams.

Filmstrips can be expressed using sequence diagrams, but then how do we add the test case information (such as where to start and what to do when it goes wrong)?

Therefore we address these issues by proposing a language extension. Where possible structural snapshots are expressed using object pattern diagrams (with OCL). As much as possible is pushed into diagrams. Filmstrips are expressed using a simple scripting language (that ultimately might be defined as the semantics of merging multiple UML behaviour diagrams).

Figure 2 is a typical information model (adapted from [6]) arising in the design of a system. A sales system consists of a contacts database, an order system, and an accounts system. Initial contacts, or *prospects*, are added by sales teams to a collection of contacts databases that constitute a CRM system. Contacts are converted into customers when they register with the company and an order system manages their commercial transactions. Orders for items and eventually fulfilled on a given date when the items are packaged up, labelled and shipped to the customer. The sales system will be used to provide examples of the use of snapshot patterns and filmstrips in the rest of this paper. Note that the association role end names default to the class names, pluralized where appropriate.

3 Snapshots

An information model describes the logical structure of information in a system. This paper proposes that the information model is used as the basis of tests by using a

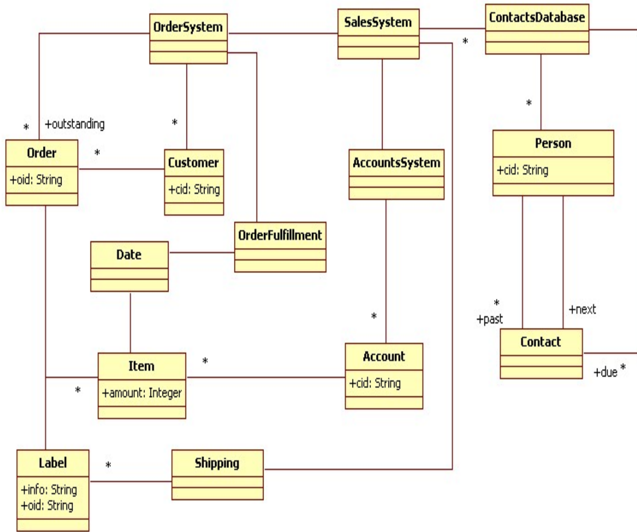

Figure 2. Information Model

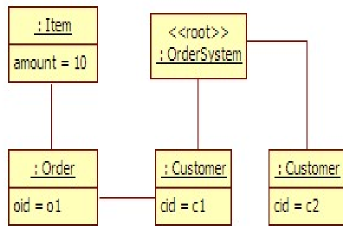

Figure 3. Initial State

mapping from snapshot patterns and filmstrips, expressed in terms of the information model, in order to generate tests on the implementation under test. This section describes the features of snapshot patterns.

A snapshot pattern is an object diagram. The object diagram must be consistent in terms of object and link types with an associated information model. A snapshot pattern that conforms to the rules of object diagrams is *ground* (meaning it contains no variables). A ground snapshot pattern can be used to describe a particular system state because the information model mapping can be used to transform it to implementation data. This is useful when creating test cases since a ground snapshot is an implementation independent description of the starting point for a test scenario.

For example figure 3 is a ground snapshot describing a sales system with two customers one of whom has an account. Notice that the snapshot labels one of the objects

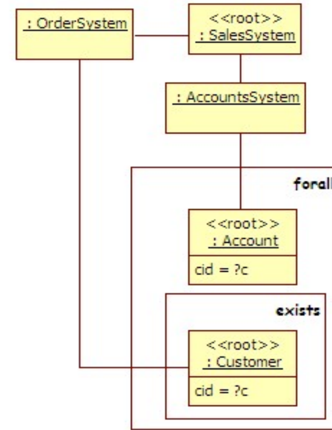

Figure 4. Accounts must have customers

«`root`». In general, each snapshot must have at least one root object. Where the root object might be ambiguous, the «`root`» label can be used.

One use of snapshots is to express system *invariants*. These are conditions that must hold at key points (generally before and after each use case) during the life-time of a system. Invariants are general rules and as such cannot be expressed using ground snapshots. Non-ground snapshots include variables. A variable may occur as the identity of an object or as the value of a slot. Variables in snapshots will be denoted as a `?` followed by a name in snapshot patterns.

Figure 4 shows an example invariant expressed using a snapshot pattern. The invariant requires that at all times if there is an account then there must be a customer with the same customer identifier. This example shows a number of features of snapshot patterns:

- quantification is expressed using a box containing a sub-snapshot pattern. Each item that is the subject of quantification is shown as a root object.
- variables may occur more than once. Each occurrence of a variable must resolve to the same value when the pattern is mapped to the implementation.
- nested boxes are used to reflec scope and t nested quantification. In the example, for each account, there must exist a customer with the same id.
- links may cross box boundaries and, in general, links (and repeated variables) can be used to show sharing of structure.

The example invariant can be expressed as follows in OCL:

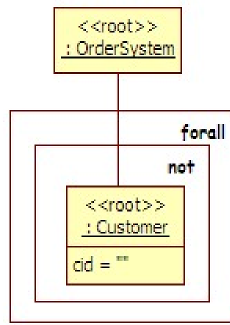

Figure 5. Customers must have Ids

```
context SalesSystem
  accountsSystem.accounts->forAll(a |
    orderSystem.customers->exists(c |
      a.cid = c.cid))
```

Whilst the semantics of the examples in the two languages are the same, OCL is often criticised by practitioners as being difficult to understand. This paper proposes that snapshot patterns make a contribution by offering precision using a notation that is familiar to modellers.

Figure 5 shows an example invariant that involves negation. It states that the identifier of each customer in the system must be a non-empty string. Consider the method that is used by a modeller when constructing this invariant. Firstly, the modeller constructs a snapshot consisting of two objects one of which is a root. The snapshot expresses an illegal configuration of objects that should be ruled out. A negation box is then drawn around the illegal object: in this case the customer with an illegal id. The snapshot is then analysed for links that are instances of associations with multiplicities other than 1. The `customers` association has a * multiplicity, therefore the snapshot pattern must have a quantification box with the customer object as its root. Since we want to make all customers conform to the snapshot pattern, a `forall` box is used.

Figure 6 shows an example of an invariant that is expressed using snapshot patterns *including* OCL expressions. The invariant states that all customers must have a unique customer identifier. The `forall` quantification box has two roots meaning that it must hold for all pairs of customers. The nested `if-then` box is a rule that states if the snapshot pattern in the `if`-part holds then the snapshot pattern in the `then`-part must also hold. OCL expressions are also snapshot patterns, in this case, OCL is used to state that of the two customers are not the same then the customer identifiers must be different.

The invariant expressed in figure 6 is a commonly occurring property of information systems. It is often the case that one or more properties of a class are unique (possibly in a given context). Figure 7 shows an example of how special syntax can be invented that captures a common pattern. Snapshot patterns can use re-write rules to define new abstractions, in this case, figure 7 is defined in terms of figure 6 with the suitable introduction of structure and variables.

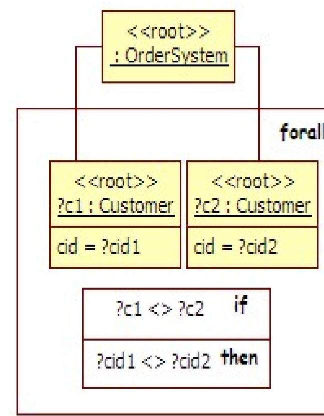

Figure 6. Unique customer identifiers

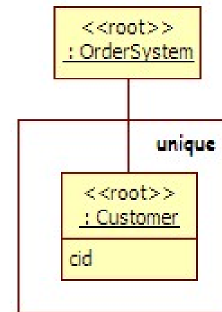

Figure 7. Special Syntax

Snapshot patterns are named and can be expressed using an alternative textual syntax. For example, the following definition corresponds to figure 4:

```
snapshot AccountsHaveCustomers {
  root object : SalesSystem {
    orderSystem = ?os
    accountSystem = ?as
  }
  object ?os : OrderSystem {
    customer = ?co
```

```
  }
  object ?as : AccountSystem {
    accounts forall {
      root object ?a : Account {
        cid = ?c
      }
      exists {
        root object ?co : Customer {
          cid = ?c
        }
      }
    }
  }
}
```

The snapshot pattern language outlines above can be used as a serialization format and is arguably easier to use in terms of semantic definitions and transformation processing.

This section has described snapshot patterns and given a collection of examples. Snapshot patterns consist of objects, with slots and links. The identifiers of objects and the values of slots can be variables. The syntax of snapshot patterns are constructively defined in terms of: OCL; objects, slots and links; quantification (universal and existential); negation; disjunction (not shown); conjunction (not shown); implication (`if-then`). New forms of snapshot patterns can be defined using rewrite rules.

4 Filmstrips

The previous section has described snapshot patterns that are used to represent logical system states. Ground snapshots are used to represent particular system states, for example the starting point for test scenarios. Non-ground snapshots contain variables and quantification and represent sets of system states. A non-ground snapshot can be used as a system invariant or to describe part of a sequence of steps performed by a system.

A sequence of steps involving pairs of logical system states is called a *filmstrip*. Each pair is called a filmstrip *step* and involves a system operation arising from a use-case. Given a mapping from a logical information model to a physical implementation, a filmstrip is a test scenario describing the expected sequence of steps performed by the system.

UML provides behavioural models (statecharts, collaboration diagrams, sequence diagrams) and OCL that can be used to express filmstrips. The information used to express filmstrips in UML is distributed amongst the various models. For example, a statechart described a system behaviour from the perspective of a single class whereas a sequence diagram describes behaviour in terms of a collection of objects. Pre and post conditions on system states are expressed using OCL.

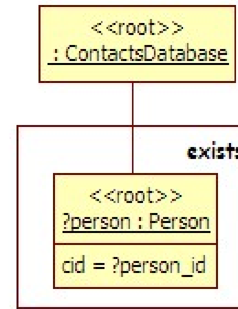

Figure 8. The PersonExists Snapshot Pattern

There is no precise semantics for denoting filmstrip behaviour in UML. Individual behaviour models do not have a semantics and the composition of the different behaviour models does not have a semantics. Furthermore, there is little support in UML for managing the complexity and supporting reuse with respect to expressing behaviour.

This paper proposes a language for expressing filmstrips in terms of snapshot patterns. The language can be given a precise semantics [4, 5] and can be used as the basis for expressing both individual and compositional behaviour of UML models. Furthermore, the filmstrip language aims to support modularity by allowing snapshot patterns to be named and reused in different contexts.

Filmstrips are defined in terms of sequences of steps. Each step consists of a named operation, its arguments, an optional return value, and a pre- and post-condition. Pre- and post-conditions are expressed as snapshot patterns. A snapshot pattern may be named and reused.

For example, consider defining a step that adds a new contact to the contacts database. Figure 8 defines a snapshot pattern that requires a person to exist. Note that both the person and the person identifier are specified as variables. Using variables for the elements in the snapshot will allow it to be used in a number of different contexts.

The filmstrip language allows snapshot patterns to be named and defined. For example:

```
let PersonExists =
      snapshot {
        root object : ContactsDatabase {
          contacts exists {
            root object ?person : Person {
              cid = ?person_id
            }
          }
      }
in ...
```

Figure 9. `addContact(c)` **pre-condition**

The body of a filmstrip expression may contain local snapshot definitions, steps and invariant specifications. An invariant is a snapshot that must hold over a given filmstrip. Assuming that the invariants from the previous section are already defined and should hold at all times:

```
always AccountsHaveCustomers and
       CustomerIDNotEmpty and
       UniqueCustomerIds {
 let PersonExists = ...
 in ...
}
```

Figure 9 shows the precondition of the `addContact(c)` operation where `c` is the supplied identifier. The pre-condition reuses the definition of `PersonExists`, but places it into a negation box which forces a person with the supplied id `c` to be absent.

Snapshot patterns can be transformed by replacing variable names in order to make operation arguments, return values, and variables in pre- and post-conditions consistent. The syntax for performing renaming is `S[n/o]` where `S` is a snapshot pattern, `n` is a new variable name and `o` is a variable name that is to be replaced in `S`.

The post-condition for `addContact(c)` simply reuses the `PersonExists` snapshot pattern and renames the variables in order to make them consistent. This can be expressed in the filmstrip language as:

```
let PersonExists = ...
in step addContact(c) {
     pre not PersonExists[?c/?person_id]
     post PersonExists[?c/?person_id]
   }
```

Now consider converting from a prospect (a person with whom the sales force has had contact) to a customer. The operation `convertContact(c)` requires that the system adds a new customer and account to the sales system. We must ensure that all the identifiers match up between the contacts database, the order system and the accounts system.

Figure 10 shows the pre-condition for `convertContact(c)`. It assumes the definition of a snapshot pattern for `AccountExists` which is equivalent to `PersonExists` (and could be defined by a suitable renaming). The pre-condition requires that there is a prospect with the supplied customer id but that there is no account with that id.

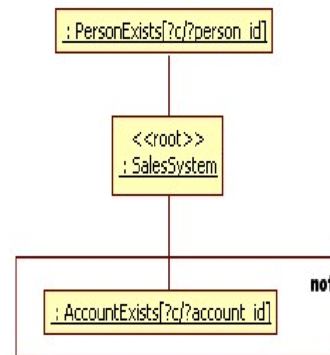

Figure 10. `convertContact(c)` **pre-condition**

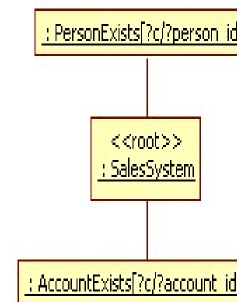

Figure 11. `convertContact(c)` **post-condition**

The post-condition for `convertContact(c)` is shown in figure 11. This requires that there is both a prospect and an account with the given id. Since we have established the invariant (see section 3) that every account must have an appropriate customer entry in the order system then the required state is achieved. The complete step can be defined as follows:

```
always ... {
  let PersonExists = ...
      AccountExists = ...
  in step addContact(c) {
       ...
     }
     step convertContact(c) {
```

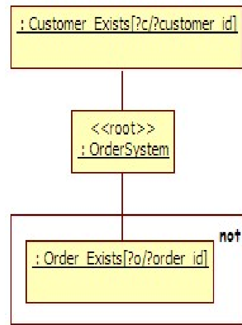

Figure 12. The `placeOrder(c)` pre-condition

```
      pre PersonExists[?c/?person_id] and
          not AccountExists[?c/?account_id]
      post PersonExists[?c/?person_id] and
           AccountExists[?c/?account_id]
    }
```

The `convertContact(c)` step is equivalent to:

```
always PersonExists[?c/?person_id] {
  step convertContact(c)
    pre not AccountExists[?c/?account_id]
    post AccountExists[?c/?account_id]
  }
}
```

Finally, consider placing an order. A customer with a given identifier places an order. The system produces an order with a unique order identifier. This example is slightly different from the previous examples since the information model shown in figure 2 requires that an order is shared between the customer and the order system. The `placeOrder(c)` operation must ensure that a customer with the supplied identifier exists and that a unique order is created.

Assuming the snapshot `OrderExists` and that the order identifier is `o` then the pre-condition for `placeOrder(c)` is shown in figure 12. The post-condition must achieve two things: it must ensure the existence of an order and the order must be shared between the customer and the order processing system. This is shown in figure 13.

Sharing is achieved in the post-condition of `placeOrder(c)` by consistently renaming the variables used to represent the object identities of the customer and the order. The final step is expressed int he filmstrip language as follows:

```
let Sharing =
```

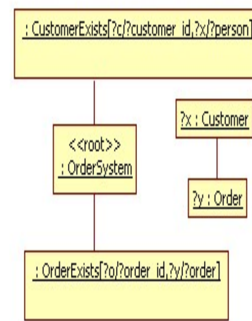

Figure 13. The `placeOrder(c)` post-condition

```
     snapshot {
       object ?x:Customer{}
       object ?y:Order{}
     }
in always CustomerExists[
            ?c/?customer_id,
            ?x/?customer] {
      step o = placeOrder(c) {
        pre = not OrderExists[?o/?order_id]
        post Sharing and
             OrderExists[?o/?order_id,
                         ?y/?order]
      }
   }
```

Notice that the identity of the new order is returned from the call of `placeOrder(c)`. This allows the pre-condition to force the absence of the order before it is required by the post-condition.

5 Analysis and Review

The AGEDIS tools for model based testing [7] uses a behavioural description that consists of statecharts and a scripting language. The proposed filmstrip language in this paper is more abstract since it does not rely on an action language to defined the effect of system operations.

[10] describes a number of approaches to model based testing that differ in terms of the type and number of models used. This work falls into the category of model based testing whereby testing models are constructed manually and differ from the design models used to construct the implementation.

OCL is used as the source of tests in a number of systems. For example [15] generate code from post-conditions

expressed as OCL. This paper argues that the use of snapshot patterns can be integrated with the modelling process and can be used to capture many aspects of tests without resorting to OCL.

Scenarios are proposed in [1] as the basis for model based testing which appears to be very similar to the approach proposed in this paper. Unfortunately, [1] is described at the proposal stage and does not contain any details of the mechanisms used to express the scenarios or the test cases. Diagrams have been proposed as the basis for constraints in [8] however these are not based on a standard notation such as UML. Visual constraint diagrams are described in [12] as the basis of system verification.

An earlier version of this work was presented as an invited talk at an ASTRANet workshop [4]. The presentation describes implementation features of the approach including a virtual machine that can be used to check snapshot patterns, a method for attaching snapshot patterns and filmstrips to Java programs and an XML format for describing test cases. The XMF system has been used to develop a domain specific language for testing that is the basis for the proposed filmstrip language in this paper [5]. The next phase in this work aims to define snapshot patterns as a profile for UML that integrates with use-cases and behaviour models.

References

[1] D. Arnold, J.-P. Corriveau, and V. Radonjic. Open framework for conformance testing via scenarios. In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion*, pages 775–776, New York, NY, USA, 2007. ACM.

[2] J. Botaschanjan, M. Pister, and B. Rumpe. Testing agile requirements models. *Journal of Zhejiang University SCIENCE*, 5(5):587–593, 2004.

[3] A. Cavarra, C. Crichton, and J. Davies. A method for the automatic generation of test suites from object models. In *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*, pages 1104–1109, New York, NY, USA, 2003. ACM.

[4] T. Clark. Iswim for testing - a model driven approach. Invited Talk, Feb 2007. Presentation available at http://itcentre.tvu.ac.uk/ clark/Presentations/ISWIM

[5] T. Clark. A domain specific language for testing. Tutorial, Feb 2008. Tutorial available at http://itcentre.tvu.ac.uk/ clark/XMF/.

[6] D. F. D'Souza and A. C. Wills. *Objects, components, and frameworks with UML: the catalysis approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[7] A. Hartman and K. Nagin. The agedis tools for model based testing. *SIGSOFT Softw. Eng. Notes*, 29(4):129–132, 2004.

[8] S. Kent. Constraint diagrams: visualizing invariants in object-oriented models. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 327–341, New York, NY, USA, 1997. ACM.

[9] A. C. D. Neto, R. Subramanyan, M. Vieira, and G. H. Travassos. A survey on model-based testing approaches: a systematic review. In *WEASELTech '07: Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies*, pages 31–36, New York, NY, USA, 2007. ACM.

[10] A. Pretschner and J. Philipps. Methodological issues in model-based testing. In *Model-Based Testing of Reactive Systems*, pages 281–291, 2004.

[11] B. Rumpe. Agile test based modeling. In *Interntional Conference on Software Engineering Research & Practice*. CSREA Press, 2006.

[12] C. J. Turner, T. N. Graham, C. Wolfe, J. Ball, D. Holman, H. D. Stewart, and A. G. Ryman. Visual constraint diagrams: Runtime conformance checking of uml object models versus implementations. *Automated Software Engineering, International Conference on*, 0:271, 2003.

[13] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing. Working Paper, April 2006. Working paper available at http://www.cs.waikato.ac.nz/pubs/wp/2006/uow-cs-wp-2006-04.pdf.

[14] M. Utting, A. Pretschner, C. M. Utting, E. Pretschner, B. Legeard, B. Legeard, M. U. A, E. P. B, and B. L. C. Abstractions for model-based testing. In *Proc. Test and Analysis of Component-based Systems (TACoS04*. Morgan Kaufmann, 2006.

[15] Šarūnas Packevičius, A. Ušaniov, and E. Bareiša. Software testing using imprecise ocl constraints as oracles. In *CompSysTech '07: Proceedings of the 2007 international conference on Computer systems and technologies*, pages 1–6, New York, NY, USA, 2007. ACM.