

Type: Research paper

Title: Testable Use Cases in the Abstract State Machine Language

Abstract: Use cases are a method for describing interactions between humans and/or systems. However, despite their popularity, there is no agreed formal syntax and semantics of use cases. The Abstract State Machine Language (ASML) is an executable specification language developed at Microsoft Research. In this paper we define an encoding of use cases in ASML and demonstrate the advantages by describing techniques to generate test cases and test oracles from the encoding.

Topics: Automated software testing, Conformance testing, Formal methods

Author: Wolfgang Grieskamp, Microsoft Research, One Microsoft Way, Redmond, WA 98052, USA, Tel. +1-425-7075740, wrrwg@microsoft.com

Author: Markus Lepper, TU Berlin, Franklinstr. 28-29, 10587 Berlin, Germany, lepper@cs.tu-berlin.de

Author: Wolfram Schulte, Microsoft Research, One Microsoft Way, Redmond, WA 98052, USA, schulte@microsoft.com

Author: Nikolai Tillmann, Microsoft Research, One Microsoft Way, Redmond, WA 98052, USA, t-niktil@microsoft.com

ISBN 0-7695-1287-9

Testable Use Cases in the Abstract State Machine Language

Wolfgang Grieskamp*

Markus Lepper†

Wolfram Schulte‡

Nikolai Tillmann§

Abstract

Use cases are a method for describing interactions between humans and/or systems. However, despite their popularity, there is no agreed formal syntax and semantics of use cases. The Abstract State Machine Language (ASML) is an executable specification language developed at Microsoft Research. In this paper we define an encoding of use cases in ASML and demonstrate the advantages by describing techniques to generate test cases and test oracles from the encoding.

1 Introduction

Abstract state machines ([8]) describe the dynamic behavior of complex systems in an intuitive but mathematically precise way. A wealth of ASM material is found at [1]. ASML is an advanced ASM-based executable specification language developed at Microsoft Research, which is intended to be used as a general notation for modeling, analyzing and rapid prototyping of components, devices and protocols. The language is fully integrated into Visual Studio and provides native COM connectivity and automation. It has been used at Microsoft to specify significant parts of device protocols and network components. A further application of ASML we envisage is to host semi-formal, domain specific notations which are mapped by transformation to the core language, this way gaining semantics and tool support.

In this paper we show one such application for ASML, presenting an encoding of *use cases* in ASML. Use cases ([10]) are a method for describing interactions between humans and/or systems which is applied in the requirements analysis phase. Despite their popularity, there is no agreed formal syntax and semantics of use cases, and thus no mechanical instrumentation for the purpose of, e.g., validating the design and the implementation w.r.t. the requirements as they are expressed by use cases.

The goals of this paper are twofold. On the one hand it is a case study on the design of ASML. On the other hand we investigate and demonstrate the benefits of a formalization of use cases by instrumenting them for *conformance testing*. We present a fully operational test

oracle as well as a test case generation algorithm, both implemented in ASML, working on our encoding of use cases. These algorithms have been validated with the implementation of ASML.

This paper extends earlier work combining use cases with Z [7]. Due to its orientation for ASML, the use case notation defined here is easier to comprehend for engineers in comparison to the Z-based one given in [7]. Regarding conformance testing, we extend work preliminary presented in [6].

2 What are Use Cases?

There is an ongoing discussion about syntax, semantics and methodology of use cases in the software engineering community (see e.g. [3]). Opposed to the graphic formalisms for *combining* use cases, e.g., by the “Use Case Diagrams” offered by UML [10], the means for specifying the *contents* of a single use case is not agreed upon at all. The UML definition just states that “a use case can be described in plain text, using operations, in activity diagrams, by a state-machine, or by other behavior description techniques...” (cited from [5]).

Typically, if use cases are given in textual form, we find documents as illustrated in Fig. 1, which represents (simplified) use cases for a cash dispenser. Note the non-determinism in the specification regarding the “bad” path for the case the card is invalid: use cases are typically loose, keeping some or many details open, which is intended feature on the level of requirements specification.

We will use the following systematic understanding of use cases, which is near to the one found in [4] and similar to the one used in [7] and in [9]:

- The systems we observe are characterized by sequences of *interactions*. Sequences of interactions are called *dialogues*.
- An interaction consists of information identifying the *actor* and the *action* performed by this actor. The actors involved in a dialogue are often one human and one technical system whose interactions alternate, but in general also several humans can talk to several machines, or machines can talk to each other. The important methodological principle is that we only look at the *observable* behavior of each actor as visible in an interaction, and that all internal state of actors is hidden.

*Microsoft Research, Redmond, wrg@microsoft.com

†TU Berlin, lepper@cs.tu-berlin.de

‡Microsoft Research, Redmond, schulte@microsoft.com

§Visiting Microsoft Research, t-niktil@microsoft.com

- Goal** user wants to draw money
State the cash supply of the dispenser
Good Path user successfully draws money
- (1) machine asks for card
 - (2) user enters card
 - (3) machine asks for amount
 - (4) user enters amount less than supply
 - (5) machine ejects card
 - (6) user takes card
 - (7) machine ejects asked amount, supply decreases
 - (8) user takes money
- Bad Path** card is invalid
- (1) machine asks for card
 - (2) user enters card
 - (3) machine ejects card
 - (4) user takes card
- Bad Path** not enough cash
- (1) machine asks for card
 - (2) user enters card
 - (3) machine asks for amount
 - (4) user enters amount greater then supply
 - (5) machine ejects card
 - (6) user takes card

Figure 1: Use Cases for a Cash Dispenser

- A use case is described by a *dialogue pattern*, which is essentially a dialogue (sequence of interactions) with some variables used to bind parameters of actions.
- We have an (observable) global system state which all use cases share. In the dialogue patterns, we can describe how this state is transformed by an interaction.

With this understanding, a set of use cases describes the set of dialogues which can be obtained by concatenating the instances of the dialogue patterns in some order. Note that we do not restrict the model to only two actors, as often found in the literature, e.g. [4]: Moreover, we do not impose *a priori* that actors in dialogues alternate.

3 A sketch of AsmL

Before we present the encoding of use cases in ASML we give a sketch of the language as far as it is needed for this paper.

ASML provides mathematical types and notations for sets, maps and sequences as they are known from text books and from pseudo-code. We will use these notations on an intuitive base throughout this paper, though we should note that they are fully formalized.

There are two key aspects which distinguish ASML from other related notations: it has a full-fledged object and

component system (with COM and COM+ integration), and it uses the ASM approach for dealing with state. We will not use the object-orientation and hence skip this part – but central for the ideas presented in this paper is the treatment of state.

State is contained in variables. An abstract state machine computes stepwise, *simultaneous* updates on these variables. When the machine executes assignments, it does not actually change the variables, but just accumulates a so-called *update-set*. If this update-set is consistent (e.g., no assignments of different values to the same variable have been queued) and the machine makes it step, the update-set is committed and the variables change their state.

Consider the following fragment of a sorting algorithm:

```

var A as Seq Integer
until fixpoint
  choose  $i \in \text{dom } A, j \in \text{dom } A$ 
    |  $i < j$  and  $A(i) > A(j)$ 
       $A(i) := A(j)$ 
       $A(j) := A(i)$ 

```

This machine performs a step in each iteration of the until fixpoint loop. The two pointwise assignments to the sequence just contribute to the update-set, which is committed in each iteration. The loop terminates when the last step has not committed any updates, thus as soon as the sequence is sorted. Note that the **choose** is non-deterministic, and hence we actually *specify* a whole family of sorting algorithms like bubble sort or quick sort.

A machine can be decomposed into sub-machines. When a sub-machine works on variables defined in the enclosing scope, it actually works on local copies, and when it terminates, it adds the computed update-set of the copies to the update-set of the enclosing sub-machine. Consider the fragment:

```

var x as Integer = 1
var y as Integer
machine
  x := x + 1
step
  x := x + 1
  y := x

```

In the context of the submachine (denoted by *machine...step...*), after the **step**, the variable x denotes 2. However, in the enclosing context, x still has its initial value 1. Thus, the update set created by the entire fragment is $x \mapsto 3, y \mapsto 1$. Note that this way side-effect freeness of computations is preserved in ASML: the order in which assignments are executed does not matter. This contributes to a clean and simple mathematical semantics of ASML, which supports building reasoning tools for it.

ASML provides an exception model similar to the one found in C++ or Java. In conjunction with sub-machines, if an exception is thrown, then the update-set of the protected block is forgotten. Consider the following fragment:

```

var x as Integer = 1
var y as Integer = 1
try
  machine
    x := x + 1
    y := y + 2
  step
    y := x + 1
    y := y + 1
catch e as CollisionException :
  skip

```

The assignments to y with the distinct values of 2 and 3 in the second step of the sub-machine cause a collision. As a consequence, an exception is thrown. On catching it, the updates produced for the variables x and y are rolled back. The possibility to arbitrarily roll-back updates is the key feature of ASML which we will use in this paper to formulate a test oracle and to generate test cases by exploration.

4 Embedding Use Cases in AsmL

It is not our intention to propose a new notation for use cases (see also Sec. 7). Instead, we envisage an adaptable refinement of existing conventions and notations by annotations with ASML. Fig. 2 shows how we may annotate and refine the informal use case specification from Fig. 1. The state is declared by an ASML variable $supply$ of type $MONEY$, which is defined as the subset of integers which are multiples of 10. The interactions performed by the user and by the dispenser, respectively, are declared next. In the paths, we then annotate each step with an interaction pattern, given as a term over an interaction and possibly free variables and constraints, as in $PutAmount(X \mid X \leq supply)$. Note that these kinds of term patterns are standard ASML. The scope of the free variables introduced this way begins at the pattern and extends until the end of the path.

The human-readable form of use-cases given in Fig. 2 is reduced to a core representation as shown in Fig. 3, which is not intended to be visible for users. Our way of encoding is as follows. A use case is given as a set of *transitions*. A transition is a procedure which takes an interaction as a parameter, possibly performs some updates on the use case's state, and updates the set of successor transitions contained in the global variable *contin*. We can view this as a (non-deterministic) automaton, where *contin* represents the automaton's control state. There is, however, one important extension compared to plain automata: the next state is calcu-

Goal user wants to draw money

State the cash supply of the dispenser:

```

type MONEY =
  {x ∈ Integer | x mod 10 = 0}
var supply as MONEY

```

Interactions user:

```

PutCard; PutAmount(X ∈ MONEY)
TakeCard; TakeMoney

```

Interactions dispenser:

```

AskCard; AskAmount
EjectCard; EjectMoney(X ∈ MONEY)

```

Good Path user successfully draws money

- (1) machine asks for card
AskCard
- (2) user enters card
PutCard
- (3) machine asks for amount
AskAmount
- (4) user enters amount less than supply
PutAmount(X \mid X ≤ supply)
- (5) machine ejects card
EjectCard
- (6) user takes card
TakeCard
- (7) machine ejects asked amount, supply decreases
EjectMoney(X) · supply := supply - X
- (8) user takes money
TakeMoney

Bad Path card is invalid

- (1) machine asks for card
AskCard
- (2) user enters card
PutCard
- (3) machine ejects card
EjectCard
- (4) user takes card
TakeCard

Bad Path not enough cash

- (1) machine asks for card
AskCard
- (2) ...

Figure 2: Annotated Use Cases

lated dynamically, and may depend on the concrete inputs as they are found in the interactions. This applies in the “good” path to the interaction $PutAmount(X)$, where the next states depend on X .

Note that in Fig. 3 we use an unusual indentation regarding nesting: the schema of an expression defining a use case is

$$\{\lambda P_1 \cdot \text{contin} := \{\lambda P_2 \cdot \text{contin} := \{\dots\}\}\}$$

```

data IACT =
cases
  PutCard; PutAmount(X as MONEY)
  TakeCard; TakeMoney
  AskCard; AskAmount
  EjectCard; EjectMoney(X as MONEY)
type MONEY = {x ∈ Integer | x mod 10 = 0}
var supply as MONEY
var contin as Set[IACT → ()]
Good = {λ AskCard ·
  contin := {λ PutCard ·
  contin := {λ AskAmount ·
  contin := {λ PutAmount(X | X ≤ supply) ·
  contin := {λ EjectCard ·
  contin := {λ TakeCard ·
  contin := {λ EjectMoney(X' | X' = X) ·
  supply := supply - X
  contin := {λ TakeMoney ·
  contin := ∅ } } } } } } } }
Bad1 = {λ AskCard ·
  contin := {λ PutCard ·
  contin := {λ EjectCard ·
  contin := {λ TakeCard ·
  contin := ∅ } } } } } }
Bad2 = {λ AskCard ·
  contin := {λ PutCard ·
  contin := {λ AskAmount ·
  contin := {λ PutAmount(X | X > supply) ·
  contin := {λ EjectCard ·
  contin := {λ TakeCard ·
  contin := ∅ } } } } } } } }
CashDispenser = Good ∪ Bad1 ∪ Bad2

```

Figure 3: Reducing the Cash Dispenser’s Use Cases

A fully automatic transformation of use cases as given in Fig. 2 to the core form in Fig. 3 is straight-forward: the interaction patterns as found in the dialogue patterns become arguments of the interaction procedures, a set of which is assigned to *contin* in each step. Note that our representation for use cases is more general as needed for the example. In the example, branching (i.e. where we assign not just a singleton set of continuation functions) is only necessary for the top-level use case as given by the value of *CashDispenser*. However, in order to optimize the encoding, we might consider to use factorization techniques when transforming the dialogue patterns to core ASML, introducing branching on arbitrary nested levels. In principle, our encoding is general enough to represent arbitrary non-deterministic automata or regular expressions of dialogue patterns.

```

var uc // contains the use case being tested
oracle(ucase, dialogue) =
  machine
    uc := ucase
    contin := ucase
  step
  try
    test(dialogue)
  return true
  catch NoMatchException :
  return false
test(d) =
  match d with
  [] : // end of dialogue
  if contin ≠ ∅
  throw NoMatchException
[a] + d' : // more interactions
  if contin = ∅
  // repetition
  machine
    contin := uc
  step
  test(d)
  else
  choose t ∈ contin | feasible(t, a, d')
  skip // found feasible transition
  ifnone // found none
  throw NoMatchException
feasible(t, a, d) =
  try
  machine
  t(a)
  step
  test(d)
  return true
  catch NoMatchException :
  return false

```

Figure 4: Test Oracle

5 Test Oracle

Fig. 4 defines a function which tests whether a given dialogue matches a use case. It is based on the procedure *test(d)* which succeeds if the dialogue *d* confirms to the use case, and otherwise throws an exception. *test* uses the function *feasible(t, a, d')* which decides whether the transition *t*, if applied to the interaction *a*, is feasible such that the remaining dialogue *d'* confirms to the use case. We use exceptions and sub-machines as explained in Sec. 3 to realize the backtracking required for the test oracle. If, in the recursive call to *test* inside of *feasible*, a dead-end is encountered, the exception *NoMatchException* will be thrown, and all updates on the control variable *contin* as well as any state variables of the concrete use case model, like the *supply* of

the cash dispenser, are undone. *NoMatchException* is thrown if a procedure is applied to a non-matching pattern (i.e. if in $t(a)$, a is not in the domain of t), or if the dialogue ends but there is a continuation expected by the use case. Note that in the case that $contin = \emptyset$, but there are still interactions, we simply reinitialize $contin$ with the use case's start transitions. This models the one or more times repetition of the behavior has described by the use case.

6 Test Generation

We present a method which produces a set of dialogues for a given use case specification, systematically covering the scenarios as given by the specification. The method is based on information which is utilized in addition to the specification:

- a finite set of actions, called the *representative actions*;
- an equivalence relation on the state of the specification, characterizing what we call *hyper states* [6].

The algorithm for test case generation computes all paths to states which can be reached using one of the representative actions. A path is terminated if a hyper state is visited for the second time.

A hyper state is a set of concrete states which are considered as equivalent. Hyper states group infinitely many states into finitely many ones. There are several possibilities to define hyper states. In [6] we characterized a hyper state as the set of those concrete states which cannot be distinguished by any of the guards of the actions of an ASM. Applied to our use case model, guards amount to the predicates we find in the dialogue patterns, such as $X \geq supply$ and $X < supply$ in the cash dispenser example. Using this approach, to decide whether two concrete states are equivalent, we simply evaluate the guards in the compared states to a vector of booleans and compare the results. For details and a theoretical discussion, see [6].

Fig. 5 defines the algorithm. We assume some abstract type *STATE* which can represent a dump of the variable assignments of our use case model. The function *getState()* extracts such a representation from the current variable assignments; the function *setState(s)* restores it. The representative actions are contained in *reprs*. Our equivalence relation is named *operator* \sim .

A test case is represented as an initial state and a test tree. A test tree branches over interactions until it reaches a terminal state. Each of the paths in the tree represents one possible run of our use case, starting from the initial state of the test case. The function *explore* calculates such a test tree, using exceptions as done in the previous section for test evaluation to model backtracking.

```

type STATE
getState() as STATE
setState(s as STATE)
reprs as Set IACT
operator  $\sim$  (s as STATE, s' as STATE) as Boolean
initialState as STATE

data TestTree =
cases
  Branch(bs  $\in$  Set[IACT  $\times$  TestTree])
  Leaf(end as STATE)
reach(Leaf(t)) = {t}
reach(Branch(ts)) =  $\bigcup\{reach(t) \mid (-, t) \text{ as } ts\}$ 

var tcases as Set[STATE  $\times$  TestTree] =  $\emptyset$ 
var working as Set STATE = {initialState}
explore() =
  if contin =  $\emptyset$ 
    throw Leaf(getState())
  else
    step
    var bs as Set[IACT  $\times$  TestTree] =  $\emptyset$ 
    foreach t  $\in$  contin, a  $\in$  reprs
      try
        step
          t(a)
        step
          explore()
      catch b  $\in$  TestTree :
        bs := bs  $\cup$  {(a, b)}
      catch NoMatchException :
        skip
    step
      throw Branch(bs)
gen(uc) =
  while working  $\neq$   $\emptyset$ 
    choose s  $\in$  working
    step
      setState(s)
      contin := uc
    step
      try
        explore()
      catch t  $\in$  TestTree :
        step tcases := tcases  $\cup$  {(s, t)}
        step working :=
          working  $\setminus$  {s}
           $\cup$  {s'  $\in$  reach(t) |
             $\neg \exists (s'', -) \in tcases \cdot s' \sim s''$ }

```

Figure 5: Test Generation

The main function of the algorithm, $gen(uc)$, computes a set of test cases using a working set of states which need to be explored. In each iteration of $gen(uc)$, we remove one state from the working set and generate a test case for it. To the working set we add those reached states (leaves) of the test case for which we have not already generated a test case with an equivalent starting state.

The termination of gen depends on whether the reachable states are actually partitioned into a finite number of hyper states. The reachable states are those which can be computed by using only the interaction representatives.

As an example, consider again the cash dispenser, Fig. 2. Define the set of interaction representatives as follows:

$\{PutCard, PutAmount(100), TakeCard, TakeMoney, AskCard, AskAmount, EjectCard, EjectMoney(100)\}$

Choose equality on the state (which is just the cash supply) for the equivalence relation. As an initial state use $supply = 150$. After flattening the test trees, we get the following four dialogues together with start and end states. Each dialogue represents one pass through the use case.

(150, [AskCard, PutCard, EjectCard, TakeCard], 150)
 (150, [AskCard, PutCard, AskAmount, PutAmount(100), EjectCard, TakeCard, EjectMoney(100), TakeMoney], 50)
 (50, [AskCard, PutCard, EjectCard, TakeCard], 50)
 (50, [AskCard, PutCard, AskAmount, PutAmount(100), EjectCard, TakeCard], 50)

7 The Larger Picture

We presented the formal encoding of use cases in ASML, and algorithms for instrumenting the encoding for test evaluation and generation. However, how the encoding from a concrete representation as given in Fig. 2 is obtained has been left open.

In fact, since the target of our efforts is the development of tools of practical usage in concrete engineering contexts, there must be a further *adaptive frontend layer* for smooth integration into existing industrial processes. We cannot suggest a particular notation from a point of research: from company to company, from application domain to application domain different standards and conventions exist how to denote use cases.

The envisaged architecture which supports the necessary flexibility is sketched in Fig. 6:

- *Tool adaptors* normalize the specific tool and document representation, converting it to XML. For example, an MS-Word document may be converted into an XML format. This representation may, however, still contain domain specific elements.

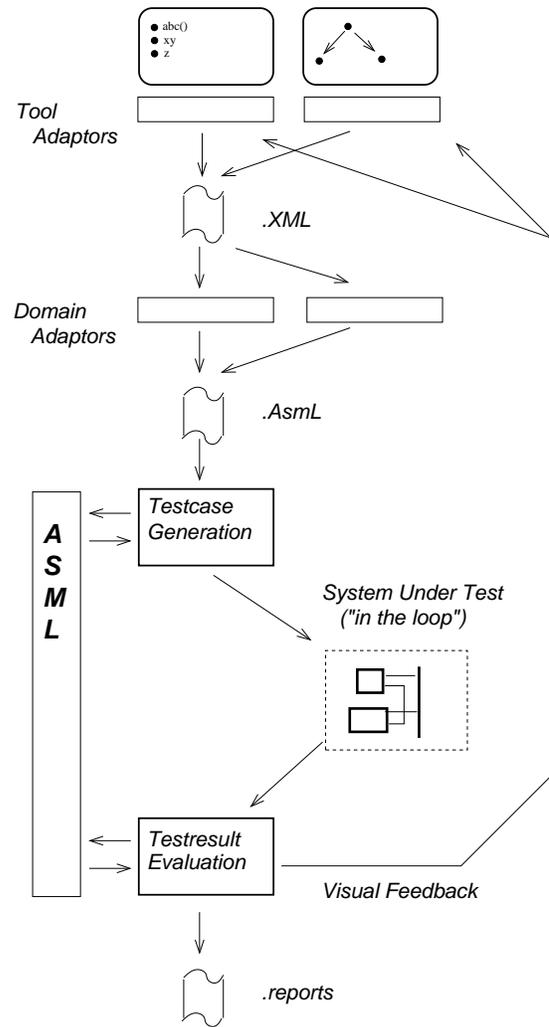


Figure 6: Tool Architecture

- *Domain adaptors* extract from the XML the actual ASML source, normalizing the domain specific parts.

Next our test generation and evaluation techniques may apply. Thereby, it is of utmost importance that we have propagated backlinks to the locations in the original sources through the transformation pipeline, since any kind of diagnostics and visual feedback has to be associated with this level.

8 Related Work and Conclusion

We have presented an encoding of use cases in the Abstract State Machine Language, ASML, and instrumented it for test evaluation and test generation, demonstrating the potential benefits a formalization of semi-formal notations can yield.

The semantic interpretation of use cases we gave follows the one we initially defined in [7]. [9] takes an almost identical approach w.r.t the basic choice of the repre-

sentation and all “political” statements are similar to ours. In contrast to [9], however, we regard the concept of an internal choice not compatible with a specification tool like use cases, and instead prefer to view branching in use cases as “angelic”. Another interesting but quite different approach to formalize use cases is found in [2], which translates the “informal meaning” of use cases into a calculus of contracts. This work aims at analyzing conditions and reasoning – breaking down the contracts to primitive state relations would yield the same semantic basis as in the other papers.

The contribution of this paper is not primarily seen in the semantic foundation and understanding of use cases, which was discussed in previous papers, but in working towards a feasible approach for the practice, which includes an notation acceptable for engineers as well as instrumentation for urgent problems in software production, like testing. To this end, we have shown that the ASML notation helps as a host language for use cases. On the one hand, it supports a style of notation similar to pseudo code which is common to most engineers and suitable for describing the state associated with a use case. Though in the toy example of the cash dispenser the state was trivial, in more real world examples convenient notations for sets, finite mappings, free data types and propositions are required, which are an expertise of ASML. On the other hand, ASML allows also to formulate the *meta* algorithms for utilizing use cases – like the test oracle and test generation algorithms we gave. The application to testing gives an important motivation for using systematic and well-founded methods in software engineering. Our approach provides a realistic scenario for black-box, conformance testing. The test generation based on use cases will remain an interactive process, since a human test engineer is still required to assign priorities to test sequences and make ad-hoc decisions like selecting representative actions; however, in comparison to classical test engineering, a much higher degree of automatization can be achieved.

REFERENCES

- 1 ASM Michigan Webpage. <http://www.eecs.umich.edu/gasm>.
- 2 Ralph-Johan Back, Luigia Petre, and Ivan Porres Paltor. Formalising UML Use Cases in the Refinement Calculus. Technical Report No. 279, Turki Center for Computer Sciene, may 1999.
- 3 Edward V. Berard. Be careful with “use cases”. Technical report, The Object Agency, Inc., 1998. http://www.toa.com/pub/use_cases.htm.
- 4 Greg Butler, Peter Grogono, and Ferhat Khende. A Z specification of use cases. In *Proc. of the Asia-Pacific Software Engineering Conference and Inter-*

national Computer Science Conference, pages 505–506. IEEE Computer Society Press, 1997.

- 5 Derek Coleman. A use case template: draft for discussion, 1998. Hewlett-Packard Software Initiative.
- 6 Wolfgang Grieskamp, Yuri Gurevich, Wolfram Schulte, and Margus Veanes. Testing with Abstract State Machines. In Roberto Moreno-Diaz and Alexis Quesada-Arencia, editors, *Formal Methods and Tools for Computer Science – EUROCAST’01 – Extended Abstracts*. University de Las Palmas, February 2001.
- 7 Wolfgang Grieskamp and Markus Lepper. Using Use Cases in Executable Z. In *ICFEM 2000 – IEEE Conference on Formal Engineering Methods*, September 2000.
- 8 Yuri Gurevich. Evolving algebras 1993: Lipari guide. In Egon Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford Univ. Press, 1995.
- 9 Perdita Stevens. On Use Cases and Their Relationships in the Unified Modelling Language. In *FASE’01*, 2001. to appear.
- 10 Uml semantics version 1.3. <http://www.rational.com/uml/index.jtimpl>.