

Making Model-Based Testing More Agile: a Use Case Driven Approach

Mika Katara and Antti Kervinen

Tampere University of Technology
Institute of Software Systems
P.O.Box 553, FI-33101 Tampere, FINLAND
firstname.lastname@tut.fi

Abstract. We address the problem of misalignment of artifacts developed in agile software development projects and those required by model-based test generation tools. Our solution is domain specific and relies on the existence of domain experts to design the test models. The testers interface the test generation systems with use cases that are converted into sequences of so called action words corresponding to user events at a high level of abstraction. To support this scheme, we introduce a coverage language and an algorithm for automatic test generation.

1 Introduction

Agile software development practices are rapidly gaining wide popularity. From the testing point of view, this can be seen as a step forward since testing is no longer seen as the last and the least respected phase of software development projects. Instead, tests are driving the development, and automated tests are replacing some design artifacts and documents. However, finding the balance between agile and plan-driven development is a difficult managerial task that needs to be made on a case-by-case basis [1].

Agile methods are lifting the status of testing by advocating automated unit and acceptance tests. However, conventional tests are limited in their ability to find defects. Test cases are scripted as linear and static sequences that help in regression testing but may prove inadequate in their coverage and maintainability. On the other hand, tests generated automatically from models can introduce the necessary variance in tested behavior or data for better coverage. Moreover, thanks to the higher level of abstraction, the maintenance of test models can be easier than test cases crafted by hand.

However, there is a mismatch between the artifacts needed in automatic test generation and those usually created in agile development projects. The models created in the latter are rarely detailed enough to be used as the basis for generating tests, since precise modeling is not seen to add any value [2]. An easy answer to this problem would be to use model-based test generation only when detailed models and heavy documentation are produced in any case, for example, in projects developing safety-critical systems.

We think that model-based testing needs to be adapted to make it more suitable for agile projects by developing highly automated and user-friendly tools. There are opportunities to introduce model-based practices to increase test coverage especially when quality is essential, for instance in development of high-volume consumer products.

Furthermore, the importance of maintainability and reusability of artifacts grows when working in a setting such as product family development. However, the right balance should be found between what should be modeled and what should not.

In our previous work [3–5] we have introduced a methodology for model-based graphical user interface (GUI) testing in the context of Symbian S60. Symbian S60 is an operating system and a GUI platform for mobile devices, such as smart phones, that has over 50 million installations [6]. Our methodology is built around a *domain-specific modeling language* consisting of so called *action words* and *keywords* [7, 8] that describe user actions at a high level of abstraction as well as their lower-level implementations, respectively. Action words are used as transition labels in labeled transition systems (LTSs) modeling the behavior of the user of the phone. They are refined to sequences of keywords with an automatic model transformation. For example, an action word for taking a picture with the phone is translated to the sequence of key strokes that conduct the action on the system under test (SUT).

Based on our experiences with the prototype implementation of our testing tools, the biggest obstacles in the practical use of our methodology concern the creation of the action word model. Moreover, even though the domain-specific language can help domain-experts without programming skills to build models, there is no clear picture on the relationship between the models created for testing, and other artifacts, especially the requirements and design documents. Another related question is about requirements coverage: “How can we know when some requirement has been covered by the generated tests?”

In this paper, we tackle the above problems. Agile development should be supported by easy-to-use tools that can show immediate added value, for instance, by finding serious defects. Thus, our approach is highly automated in a way that all the complexity associated with model-based test generation is hidden from the tester. To achieve the high level of automation, the approach is domain-specific. However, a similar approach can be developed also for other domains.

In more detail, we build our solution on use cases. Since our domain is quite restricted, experts can handle the actual test modeling while testers define test objectives based on use cases. On the one hand, use cases should be familiar to most system level testers. On the other hand, use cases can help us to solve one re-occurring problem in model-based testing, i.e., how to restrict the set of generated tests.

The structure of the remainder of the paper is as follows. In Section 2 we introduce the background of the work. Our contributions on adapting model-based testing to agile development are presented in Section 3. This includes the definition of an improved coverage language that can be used in use case driven testing and the associated algorithm for test generation. Related work is presented in Section 4 and Section 5 concludes the paper with some final remarks.

2 Background

To present the background of our work briefly, we first review the role of requirements in agile software development. Second, we describe our domain-specific test automation methodology including the so-called 3-tier test model architecture [4].

2.1 Role of Requirements in Agile Development

Requirements form the foundation upon which the entire system is built. One of the most important roles of testing is to validate that the system meets its requirements. However, we see the role of requirements in agile development quite different from conventional plan-driven projects. In the latter case requirements are documented in the beginning of the project. Any changes to requirements in later stages of the development should be avoided, due to the high costs involved in rework. In the former case, changes to requirements are considered to be inevitable. In XP [9] for instance, simple descriptions of requirements are documented as “user stories” that are used mainly for release planning. The actual implementation should be based on the face-to-face interaction with the customer or the end user. While the costs of changes can also be high in agile development, the increased customer satisfaction is seen as more important and also more cost effective in the course of the whole project.

Agile Modeling [2] is an extension to more code-centric agile methods, such as XP. It suggests developing models that can help to design and document the implementation, quite similarly to plan-driven approaches but emphasizing certain agile principles and light weight. Requirements can be captured using UML like diagrams or simple textual format, for instance.

Use cases are one of the most popular methods of capturing requirements in industrial projects. Whether textual or visual, they can be written with various levels of formality, depending on the criticality of the system under development. However, in Agile Modeling, the use cases should contain just enough information to communicate the idea and no more. Instead of developing all the use cases in the beginning, it is suggested that they should be defined in iterations and refined incrementally when needed just like other artifacts developed during the project.

The definition of a use case varies in the literature. Here we will use the informal use case format, as defined in [2]. It includes three elements: name, identifier and the basic course of action, which is a sequence of event descriptions corresponding to the high-level interaction between the user and the system. We will use the following informal use case as a running example:

Name: Alice asks Bob to meet for lunch

Identifier: UC1

Basic course of action:

1. Alice sends Bob an SMS asking him to meet for lunch.
2. Bob replies by sending a multimedia message containing a picture of a bumper taken at a traffic jam and a text “I will call you when I’ll get there”.
3. After a while, Bob calls to Alice, but she is talking to Carol so the line is busy.
4. After a few minutes Alice calls back to Bob and they agree to meet at the cafeteria.

All use cases are not equally important. In order to cope with this, some risk-based testing practices [10] should be deployed: the use cases must be prioritized based on some formal risk analysis or customer intuition. From the testing point of view, the source of the priority is less important; what is important is that the tests should be executed in the priority order. This way it can be ensured that at least the most important

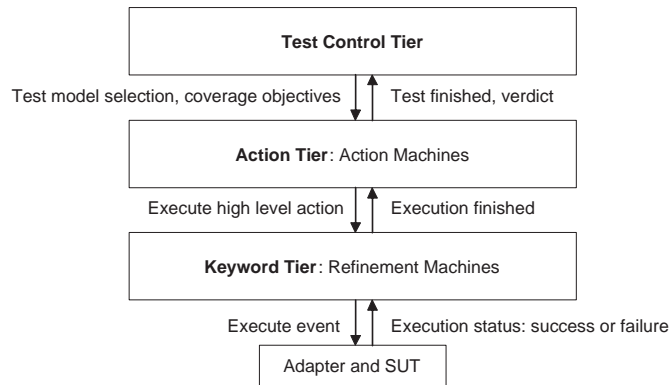


Fig. 1. 3-tier model architecture.

risks have been covered if testing needs to be discontinued for some reason. However, because there is always a risk that the risk analysis has been done poorly, it is suggested to aim at covering each use case at least in some detail.

2.2 Domain-Specific Test Modeling

Our approach to domain-specific test modeling using keywords and action words is built around the 3-tier model architecture, consisting of *control*, *action* and *refinement machines* each in its own tier (see Figure 1). The machines are LTSs that model the behavior of the user of the SUT. In order to support testing of different products of the same product family, a high-level functionality is separated from the GUI events. This facilitates the reuse of action machines with SUTs that support the same operations on different kind of GUIs. Refinement machines in the Keyword tier are used for refining the action words in action machines to sequences of executable events in the GUI of the SUT. To obtain an executable test model, we compose action machines with their corresponding refinement machines, i.e., the machines on the two lowest tiers in the model architecture.

For execution, we use an *on-line* approach, i.e., we run tests as we generate them. The tests are executed using a commercial Windows GUI test automation tool running on a PC and an adapter tool connecting the PC to the SUT. A separate log file is used for re-running the test when, for example, debugging.

In the following, the three tiers are presented (see [4] for more details).

Keyword tier In Figure 2 $\text{Camera}_{\text{RM}}$ is a refinement machine for a Camera application. In its initial state (the filled circle) the machine refines high level actions for starting the application (awStartCam) and for verifying that the application is running (awVerifyCam).

Keywords are used for generating input events to the SUT as well as making observations on the SUT. A test oracle is encoded in the model. During the test run, execution

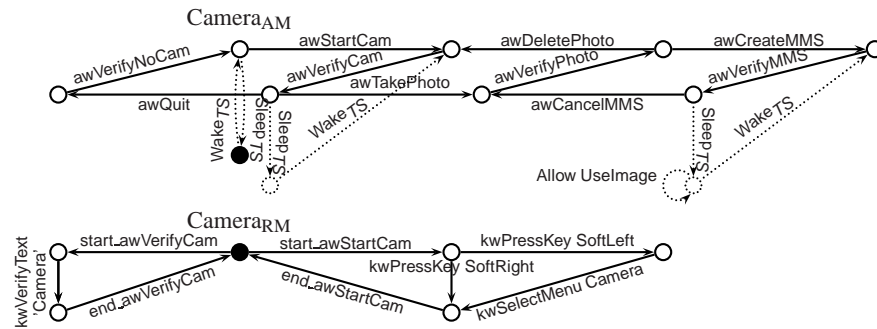


Fig. 2. Action machine and refinement machine.

of a keyword always either succeeds or fails. For example, executing `kwVerifyText 'Camera'` succeeds if string “Camera” is found on the display and fails otherwise. Sometimes failure is allowed or even required. Allowed results are expressed in the labels of transitions; an error is found if the status of an execution does not match any of these.

In a test model library, the keyword tier consists of a number of machines refining several action words. Each of the refinement machines interacts only with one action machine. Usually, the refinement corresponds to a simple macro expansion: an action word is always implemented with the same sequence of keywords. However, in some cases the sequence may vary depending on the action words executed earlier. For example, the keyword implementation of “activate Camera application” could be different, depending on whether or not the application is already running in the background.

Action tier Action machines model concurrently running applications. The machines contain action words whose executions can be interleaved to the extent defined in this tier. In a Symbian S60 GUI context, it is sufficient to define few dozens of keywords. However, the number of action words is required to be much higher.

Interleaving the executions of the action machines is an essential part of our domain-specific modeling approach. Symbian S60 applications should always be interruptible: user actions, alarms, received phone calls and messages may stop the ordinary execution of the application at any time. Obviously, it is hard for developers to ensure that applications behave well in every case. The number of cases that should be tested is far beyond the capabilities of conventional testing methods, whether automated or not. To facilitate the creation of test models where the action machines are automatically interleaved, the concepts of running and sleeping action machines have been introduced.

There are two kinds of states in action machines: running and sleeping states. An action word can be executed only when the corresponding machine is running, i.e., it is in a running state. The interleaving mechanisms guarantee that there is always exactly one action machine in a running state. In the beginning of a test run, the running action machine is a special task switcher.

`CameraAM` in Figure 2 is a simple action machine for testing the Camera application. The machine tests the following functionality: starting the camera application (`awStartCam`), taking a picture (`awTakePhoto`), and creating a multimedia message con-

taining the picture (`awCreateMMS`). The three dotted states are sleeping states, the left-most of which is the initial state. The application is started when the machine wakes up for the first time. After that, it is verified that the application actually started. Then, the test guidance algorithm makes a choice between taking a photo, quitting the application, and entering a sleeping state.

At this tier, there are two communication mechanisms between the machines. The first one controls which action machine is currently running using primitives `SleepTS`, `WakeTS`, `SleepApp` and `WakeApp`. These represent putting to sleep and waking up an application with a task switcher (*TS*) or directly within another application (*App*). The other mechanism is for exchanging information on shared resources between the machines: there are primitives for requesting (`Req`) and giving permissions (`Allow`). The former can be executed only between running states and the latter between sleeping states.

Test control tier The test control tier is used for defining which test models to use and what kind of tests to run in which order. Naturally, there are different needs depending on whether we are performing a quick test in conjunction with a continuous integration [11] cycle or chasing the cause for some randomly appearing strange behavior.

We have initially identified three different testing “modes” that should be supported in an agile project. Firstly, smoke tests are needed for verifying that a build has been successful. Such verification step can be included in the test run for each continuous integration build, for instance. However, we do not restrict to a static sequence of tests such as in conventional test automation. Instead, we may set limits on the duration of the test and explore the test model on a breadth-first fashion within those limits.

Secondly, we need to be able to cover certain requirements. Goals for testing projects are often set in terms of requirements coverage; for example, at least requirements R1, R2 and R4 should be tested. Thirdly, we would like to do serious bug hunting. In this case our primary motivation is not to cover certain requirements or to stop within five minutes. Instead, we try to find as many defects as possible. However, requirements can be used to guide the test generation also in this mode. Furthermore, the coverage data obtained in the previous test could be used to avoid retesting the same paths again.

It is also in this tier where we define which machines are composed into an executable test model. Obviously, composing all the machines is usually not necessary. In cases where we use requirements to limit the test generation, we include in the composite model only those machines that are needed to cover the requirements. However, in smoke testing mode, the machines required to be composed should be explicitly stated: machines for the Camera, Messaging and Telephony applications, for example. Moreover, within the Symbian S60 domain, the task switcher machine is always included in the executable test model. Since the state-space explosion can occur when composing test models by interleaving executions of a number of machines, instead of generating the composite test model at once, we do the parallel composition on the fly.

3 From Informal Use Cases to Model-Based Tests

This section presents the process of use case driven test generation in conjunction with the associated coverage language and test generation algorithm.

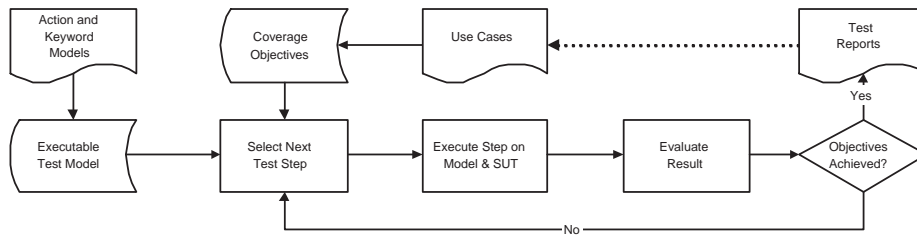


Fig. 3. Use case driven testing.

3.1 Use Case Driven Testing

Organizations developing software in an agile manner prefer testing tools that are easy to use, can find defects effectively, and integrate seamlessly with the agile processes. In the following we will concentrate on the first and the last of these requirements. Firstly, the sophisticated algorithms etc. should be hidden as much as possible from the tool users. Secondly, the input of the tools should be something that is produced in the project anyway.

We think that it is hard to get average testers to build test models. Thus, we developed a visual and easy-to-grasp domain-specific language comprising LTSs and action words. However, in a strict domain such as ours, the test models themselves could be developed by a third party. Alternatively, the adapting organization could train one or two dedicated experts to build test models using requirements, design documents, etc. In both cases, the basic test model library consisting of the fundamental models, such as machines for calling, contacts, calendar and camera in our case, may be developed with a reasonable effort. Moreover, the library could be extended incrementally based on the new requirements to be tested.

We suggest that the testers should primarily interact with the test automation system through coverage objectives that drive the test generation. To achieve this, we must link the informal use cases to coverage objectives and define test generation algorithms based on the objectives. Moreover, there should be a clear mapping between the use cases and the tests; testers need to be able to report test coverage in terms of use cases. This scheme is depicted in Figure 3.

As discussed in Section 2.1, we make the following assumptions about the requirements used as input to the test automation:

1. The requirements are stated as informal use cases including a name, an identifier and the basic course of actions.
2. The uses cases have been prioritized, for instance, based on a risk analysis.

The tester maps the events listed in the basic course of actions to action words. For instance, a spreadsheet can be used to list the action words corresponding to each event. As discussed above, if there are no predefined action words corresponding to some event, the test model library needs to be extended with new models. For traceability and

Related use case: Alice asks Bob to meet for lunch

Use case identifier: UC1

Action word sequence:

1. Alice.Messaging.awCreateSMS “Would you like to meet for lunch?”
2. Bob.Messaging.awReadSMS
3. Bob.Camera.awTakePhoto
4. Bob.Messaging.awCreateMMS “I will call you when I’ll get there”
5. Alice.Messaging.awReadMMS
6. Alice.Contacts.awSelectCarol
7. Carol.Telephone.awAnswerCall
8. Bob.Contacts.awSelectAlice
9. Bob.Contacts.awVerifyBusy
10. Carol.Telephone.awHangUp
11. Alice.Contacts.awSelectBob
12. Bob.Telephone.awAnswerCall
13. Bob.Telephone.awHangUp

Fig. 4. An action word sequence based on use case UC1

comprehensibility, the action word sequence is annotated with the use case name and identifier. The priority affects the test guidance, so it must be stated as well.

Based on the above, our running example would result in the action word sequence presented in Figure 4. The action words are chosen from a model consisting of three actors: Alice, Bob, and Carol. When running in the requirements coverage mode, for instance, the sequence in Figure 4 can be automatically expanded based on the models referenced in the names of the action words. An expanded sequence is presented in Figure 5. From the intermediate steps the tester can make sure that there is a sensible way to execute the sequence in the model.

- | | |
|--|--|
| <ol style="list-style-type: none">1. Alice.Messaging.awCreateSMS <i>Msg1</i>
Alice.Messaging.awVerifySMS
Alice.Messaging.awOpenRecipientList
Alice.Messaging.awChooseBob
Alice.Messaging.awSendSMS2. Bob.Messaging.awReadSMS
Bob.Camera.awStartCam
Bob.Camera.awVerifyCam3. Bob.Camera.awTakePhoto4. Bob.Messaging.awCreateMMS <i>Msg2</i>
Bob.Messaging.awVerifyMMS
Bob.Messaging.awOpenRecipientList
Bob.Messaging.awChooseAlice
Bob.Messaging.awSendMMS5. Alice.Messaging.awReadMMS | <ol style="list-style-type: none">6. Alice.Contacts.awOpenAddressBook
Alice.Contacts.awSelectCarol
Alice.Contacts.awDialSelected7. Carol.Telephone.awAnswerCall
Bob.Contacts.awOpenAddressBook8. Bob.Contacts.awSelectAlice
Bob.Contacts.awDialSelected9. Bob.Contacts.awVerifyBusy10. Carol.Telephone.awHangUp11. Alice.Contacts.awSelectBob
Alice.Contacts.awDialSelected12. Bob.Telephone.awAnswerCall13. Bob.Telephone.awHangUp |
|--|--|

Fig. 5. A detailed action word sequence generated from the model

3.2 Coverage Language

The action word sequences are processed further using the coverage language that will be introduced next. The design of the language has been guided by the following principles:

1. *Syntax should be concise and readable.*
2. *Elements can be required to be covered in a free order or in some specific order.*
3. *There can be alternative coverage criteria.*
4. *Criteria can relate to both test environment and test model.* A test criterion can be fulfilled, for example, if the test run has already taken too long, if some resources in the test system are running low, or if some elements in the test model are covered in sufficient detail.
5. *Execution paths in the test model must not be restricted by the language.* We keep the roles of the coverage criteria and the test model separate. The test model (alone) specifies what can be tested, whereas the coverage criteria specifies the stopping condition for test runs.

A coverage criterion (*CC*) is either an elementary criterion (*EC*) or a combination of two other coverage criteria:

$$CC = EC \mid (CC \text{ (and \mid or \mid then) } CC)$$

The operators that combine the criteria have the following meaning

- and** requires that both coverage criteria are fulfilled in any order and or simultaneously. (Design principle 2, free order.)
- or** requires that at least either one of the criteria is fulfilled. (Design principle 3.)
- then** requires that the second criterion is fulfilled after the first one. (Design principle 2, a specific order.)

Note that **A then B** does not require that *B* must not be fulfilled before *A*, it only requires that *B* is fulfilled (possibly again) after fulfilling *A*. The reasons for this will be elaborated at the end of the section.

Elementary criteria consist of two parts, a query and a requirement for the return value of the query:

$$\begin{aligned} EC &= Req \text{ for } Query \\ Req &= (\text{every} \mid \text{any}) \text{ value} \geq n \\ Query &= (\text{actions} \mid \text{states} \mid \text{transitions} \mid \text{sysvars}) \text{ regexps} \end{aligned}$$

The query part of an elementary criterion returns a set of item-value pairs. The items are actions, states, transitions or test system variables. While the first three items relate to the test model, the test system variables give access to time and date, amount of free memory in the SUT and the number of executed actions, for instance. There is an item-value pair for every item that matches any regular expression in the query.

The meaning of the value associated with an item depends on the type of the item. For actions, states and transitions we use the number of times the item has been executed

(actions and transitions) or visited (states) during the test run. For system variables it is natural to choose the value of the variable to occur in the pair.

There are two quantifications for the values in the return value set. Either *every* or *any* value is required to satisfy the condition. Once the requirement is met, the elementary criteria is fulfilled.

We often use the coverage language for setting coverage requirements based on the action word sequences. For that purpose, we define a short-hand notation (design principle 1). If the requirement part of a query is omitted, it defaults to “**every value** ≥ 1 ” or “**any value** ≥ 1 ” depending on whether or not the type of items is given in the plural form. For example, the requirement that SendMMMessage and SendShortMessage are tested after making a call can be written as follows:

action MakeCall **then actions** Send. *Message

which is a short-hand notation for

any value ≥ 1 **for actions** MakeCall
then
every value ≥ 1 **for actions** Send. *Message

The use case of our running example would be converted to the following coverage language sentence simply by adding “**action**” in front of every action word and joining the results with “**then**”:

action Alice.Messaging.awCreateSMS
then action Bob.Messaging.awReadSMS
then action Bob.Camera.awTakePhoto
then action Bob.Messaging.awCreateMMS
then action Alice.Messaging.awReadMMS
then action Alice.Contacts.awSelectCarol
then action Carol.Telephone.awAnswerCall
then action Bob.Contacts.awSelectAlice
then action Bob.Contacts.awVerifyBusy
then action Carol.Telephone.awHangUp
then action Alice.Contacts.awSelectBob
then action Bob.Telephone.awAnswerCall
then action Bob.Telephone.awHangUp

The data in the use case (the messages “Would you like to meet for lunch?”, “I will call you when I’ll get there”) is stored in a separate table which is used as a data source in the test run.

Coverage requirements can also be built from a number of use cases. Both **and** and **then** operators are sensible choices for joining the requirements obtained from the use cases. However, depending on the guidance algorithm, they may result in very different test runs. But before we can show why, we have to show how the operators affect the evaluation of the coverage criteria.

Next we define an evaluation function f that maps coverage criteria to real numbers from zero to one. The number describes the extent to which the criterion has been

fulfilled, number one meaning that the criterion has been completely fulfilled. The evaluation function is defined using function E , which evaluates an elementary criterion to a real number, and three $\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ functions. The three functions are T for evaluating **and**, S for **or**, and R for **then**:

$$\begin{aligned} f(EC) &= E(EC) \\ f(A \text{ and } B) &= T(f(A), f(B)) \\ f(A \text{ or } B) &= S(f(A), f(B)) \\ f(A \text{ then } B) &= R(f(A), f(B)) \end{aligned}$$

Let us first consider function E for elementary criteria. Once an elementary criteria EC has been fulfilled, $E(EC) = 1$. Defining that otherwise $E(EC) = 0$ would be easy and it would not contradict the semantics. However, the evaluation would not give any hint for test guidance algorithms about smaller advances in the coverage. When the search depth of the guidance algorithms is bounded, or a best-first [12] search is used, a more fine-grained evaluation is useful.

Of course, it is not always possible to give more detailed information than 0 and 1. For instance, if a coverage requirement requires that a single state in the test model has been visited at least once, we are clearly dealing with 0 or 1 value. On the other hand, a requirement that every action in a set of actions is executed at least once can be thought to be one step closer to the fulfillment every time a new action in the set is executed. More generally, consider criterion EC that sets an upper limit to a monotonically growing query value. $E(EC)$ evaluates closer to 1 every time the result of the query grows, reaching 1 when the result reaches the limit.

Based on this, we define E as follows for elementary criterion $EC = Req$ for $Query$:

$$E(EC) = \begin{cases} \min(n, \max(Query))/n & \text{if } Req \text{ is } \mathbf{any\ value} \geq n \\ avg_w_ulimit(Query, n) & \text{if } Req \text{ is } \mathbf{every\ value} \geq n \end{cases}$$

where avg_w_ulimit is the average of values returned by the query so that values greater than n are replaced by n . Thus $E(\mathbf{every\ value} \geq 1 \text{ for actions } a \text{ } b) = 0.5$ when a has been executed three times and b has not been executed.

Next, we define the evaluation function so that logically equivalent coverage criteria produce equal values. That is, the evaluation function respects the idempotence (1.x), symmetry (2.x), associativity (3.x) and distributivity (4.x) laws presented in Table 1.

A basic result for norms in fuzzy logic [13] says that the only functions that satisfy the properties x.1 and x.2 in Table 1 are the following:

$$\begin{aligned} f(A \text{ and } B) &= T(A, B) = \min(f(A), f(B)) \\ f(A \text{ or } B) &= S(A, B) = \max(f(A), f(B)) \end{aligned}$$

When T and S are defined as above, the following R satisfies properties 4.3 and 4.4:

$$f(A \text{ then } B) = R(A, B) = \frac{f(A) + f(B_{\text{after } A})}{2}$$

Where $B_{\text{after } A}$ denotes coverage requirement B whose covering does not start until A has been covered. Therefore, $f(B_{\text{after } A}) = 0$ if $f(A) < 1$.

$$\begin{aligned}
f(A \text{ and } A) &= f(A) & (1.1) \\
f(A \text{ or } A) &= f(A) & (1.2) \\
f(A \text{ and } B) &= f(B \text{ and } A) & (2.1) \\
f(A \text{ or } B) &= f(B \text{ or } A) & (2.2) \\
f((A \text{ and } B) \text{ and } C) &= f(A \text{ and } (B \text{ and } C)) & (3.1) \\
f((A \text{ or } B) \text{ or } C) &= f(A \text{ or } (B \text{ or } C)) & (3.2) \\
f(A \text{ and } (B \text{ or } C)) &= f((A \text{ and } B) \text{ or } (A \text{ and } C)) & (4.1) \\
f(A \text{ or } (B \text{ and } C)) &= f((A \text{ or } B) \text{ and } (A \text{ or } C)) & (4.2) \\
f(A \text{ then } (B \text{ and } C)) &= f((A \text{ then } B) \text{ and } (A \text{ then } C)) & (4.3) \\
f(A \text{ then } (B \text{ or } C)) &= f((A \text{ then } B) \text{ or } (A \text{ then } C)) & (4.4)
\end{aligned}$$

Table 1. Equal coverage criteria

Let us now get back to the two possible ways to join use cases to a single coverage requirement. Assume that we have converted two use cases to coverage requirements CC_{UC1} and CC_{UC2} . They can both be tested at once by combining them to the requirement “ CC_{UC1} **and** CC_{UC2} ” or “ CC_{UC1} **then** CC_{UC2} ”. Our test guidance algorithm is a greedy bounded-depth search, which will be presented in the following subsection. Here it is enough to know that the algorithm chooses the path of at most length d (the search depth) where the value evaluated for the coverage requirement is maximal. Using the **then** operator to combine the requirements implies a test run where the first use case is fulfilled before the second. But if the use cases are combined with the **and** operator, it would result in a test run where the execution of the use cases advances roughly side-by-side. Thus, we suggest combining the use cases with the same priority using **and** and use cases with different priorities with **then**.

We have excluded the negation “**not**” in the language because of the design principle 5. Consider the following (false) example which would state that making a phone call should be tested without sending an email at any point of the test run:

action MakeCall **and not action** SendEmailMessage

The negation would provide a way to restrict the behavior of the test model. Therefore, the language would not anymore state the coverage criteria; it would also change the test model. This would break the separation between the roles of the test model and the coverage language (design principle 5). The same applies also to many other operators that we considered. For example, stating that something should be tested strictly before something else is tested breaks the same principle. This is the reason for the limitations of our “**then**” operator.

In the language, there is a nice property obtained from the design principle 5 together with our requirement that the initial states of test models are reachable from every other state of the models. Consider finite sets of coverage criteria that can be fulfilled in the same test model and that talk only about the model elements (actions, states and transitions), not system variables. Every new criterion build by combining the criteria in any set with **and**, **or** and **then** operators can also be fulfilled in the same test model. This gives the testers the freedom to choose any combination of valid coverage criteria to define the stopping condition for the test run.

```

NextStep(s : state, depth : integer, c : coverage requirement)
1 if depth = 0 or s.outTransitions() =  $\emptyset$  or c.getRate() = 1 then return (c.getRate(),  $\{\}$ )
2 best_rate = 0; best_transitions =  $\{\}$ 
3 for each t  $\in$  s.outTransitions() do
4   c.push()
5   c.markExecuted(t)
6   (new_rate, dont_care) = NextStep(t.destinationState(), depth - 1, c)
7   c.pop()
8   if new_rate > best_rate then best_rate = new_rate; best_transitions =  $\{t\}$ 
9   if new_rate = best_rate then best_transitions = best_transitions  $\cup$   $\{t\}$ 
10 end for
11 return (best_rate, best_transitions)

```

Fig. 6. An on-line test guidance algorithm using the coverage requirements

3.3 Using Coverage Language in Test Generation

Next, we will present a simple algorithm that can be used in test generation. First, let us describe briefly the data structure that we use for storing and evaluating coverage requirements in our on-line test generation algorithms.

Parsing a coverage requirement results in a tree where leaf nodes represent elementary requirements and the other nodes the operators **and**, **or** and **then**. Every node implements *markExecuted(transition)* method. **And** and **or** nodes pass the calls to all children, **then** nodes pass the call to the first child that has not been fulfilled yet (see the definition for the *R* function), and leaf nodes update their item execution tables. The tables, indexed by the queried items, store the number of executions/visitations of each item.

All nodes also offer *push()* and *pop()* methods whose calls are always passed through the tree to the leaf nodes. The leaf nodes either push or pop the current execution tables to or from their table stacks. Push and pop methods allow guidance algorithms to store the current coverage data, evaluate how the coverage would change if some transitions were executed, and finally restore the data.

Lastly, there is *getRate()* method which returns the fulfillment rate of the requirement represented by the node and its children. Non-leaf nodes calculate the rate by asking first the rates of their child nodes and then using *T*, *S* or *R* function, depending on the operator of the node. Leaf nodes evaluate the value with the *E* function.

The *NextStep* function, presented in Figure 6, can be used as a core of the test generation algorithms. Three parameters are given to the function: a state from which the step should be taken, the maximum search depth in which the algorithm has to make the decision, and the coverage requirement, that is, the root node of the tree. The function returns a pair: the best rate that is achievable in the given number of steps (search depth) and the set of transitions leaving the given state. Any of those transitions can be executed to achieve the coverage rate in the given number of steps.

For simplicity, we did not take into account in *NextStep* that the SUT may force the execution of certain transitions in some states. Instead, we assumed that the suggested transition can always be executed. If this is not the case, one can use the expectation

value for the fulfillment rate in place of *getRate()* (for details, see “state evaluation” in [14]).

4 Related Work

The idea of using use cases (or sequence diagrams) to drive test generation is not new. In addition, it has been suggested to use more expressive formalisms, such as state machines [15, 16]. Traceability between requirements and model-based tests has also been studied before. For instance, Bouquet et al. present an approach in [17] where the idea is to annotate the model used for test generation with requirement information. The formal model is tagged with identifiers of the requirements allowing model coverage to be stated in terms of requirements. This allows automatic generation of a traceability matrix showing relations between the requirements and the generated test suite.

We have tackled these issues from a slightly different angle. In a restricted domain such as ours, test modeling can be assigned to some internal experts or third parties. Then, the primary task of the test automation engineer is to transform use cases to sequences of predefined actions words. The action words correspond to concepts familiar to testers in the particular domain, thus facilitating the translation. Our generation algorithms use the action word sequences as coverage objectives. Moreover, a test run produces a test log that can be used for generating reports based on requirements coverage.

In formal verification, the properties concerning models are commonly stated in terms of temporal logics such as LTL [18] and CTL [19]. This approach has been adopted also for test generation in model-based testing, for example in [20]. We defined a simpler and less expressive language for coverage for two reasons. Firstly, using temporal logics require skills not too often available in testing projects. Secondly, the strength of logics is great enough to change (restrict) the behavior of the test model: fulfilling a criterion can require that something is not tested. What we gained is that coverage requirements cannot conflict.

5 Conclusions

In this paper we have introduced an approach to adapting model-based testing practices in organizations developing software using agile processes. Such organizations are often reluctant to develop detailed models needed in most other approaches. Our approach is based on a domain-specific methodology that entails high-level of automation. The test models are developed incrementally by internal experts or third parties, and the informal uses cases are used to drive the test generation. This involves simple translation from the events listed in the use cases to actions words used in the high-level test models. Such action words describe the abstract behavior that is implemented by lower-level keyword models in the test model library.

We have also defined a test coverage language used in producing coverage objectives from the sequences of action words. The language supports different kinds of testing modes such as requirements coverage, bug hunting, or smoke testing. In the first

two modes, the coverage objectives obtained from the use cases are used as input to the test generation algorithm.

We are currently implementing a tool set supporting our scheme. The future work includes conducting industrial case studies to assess the overall approach as well as investigating the defect-finding capability of the presented heuristic. Moreover, since model-based tests include complex behavior, some of the defects can be very hard to reproduce. Towards this end we must explore different possibilities to ease debugging.

Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful comments.

References

1. Boehm, B., Turner, R.: *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison Wesley (2004)
2. Ambler, S.W.: Agile modeling homepage. Available at <http://www.agilemodeling.com> (2006)
3. Kervinen, A., Maunumaa, M., Pääkkönen, T., Katara, M.: Model-based testing through a GUI. In: *Proceedings of the 5th International Workshop on Formal Approaches to Testing of Software (FATES 2005)*, Edinburgh, Scotland, UK, Number 3997 in LNCS, Springer (2006) 16–31
4. Kervinen, A., Maunumaa, M., Katara, M.: Controlling testing using three-tier model architecture. In: *Proceedings of the Second Workshop on Model Based Testing (MBT 2006)*, ENTCS **164(4)** (2006) 53–66
5. Katara, M., Kervinen, A., Maunumaa, M., Pääkkönen, T., Satama, M.: Towards deploying model-based testing with a domain-specific modeling approach. In: *Proceedings of TAIC PART - Testing: Academic & Industrial Conference*, Windsor, UK, IEEE Computer Society (2006) 81–89
6. S60: Symbian S60 homepage. Available at <http://www.s60.com> (2006)
7. Fewster, M., Graham, D.: *Software Test Automation*. Addison-Wesley (1999)
8. Buwalda, H.: Action figures. *STQE Magazine*, March/April 2003 (2003) 42–47
9. Wells, D.: Extreme programming: a gentle introduction. Available at <http://www.extremeprogramming.org> (2006)
10. Craig, R.D., Jaskiel, S.P.: *Systematic Software Testing*. Artech House (2002)
11. Fowler, M.: Continuous integration. Available at <http://www.martinfowler.com/articles/continuousIntegration.html> (2006)
12. Russel, S., Norvig, P.: *Artificial Intelligence*. Prentice-Hall (1995)
13. Klement, E.P., Mesiar, R., Pap, E.: *Triangular Norms*. Springer (2000)
14. Kervinen, A., Virolainen, P.: Heuristics for faster error detection with automated black box testing. In: *Proceedings of the Workshop on Model Based Testing (MBT 2004)*, ENTCS **111** (2005) 53–71
15. Jard, C., Jéron, T.: TGV: theory, principles and algorithms – a tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *STTT* **7** (2005) 297–315
16. AGEDIS Consortium: AGEDIS project homepage. Available at <http://www.agedis.de/> (2004)

17. Bouquet, F., Jaffuel, E., Legiard, B., Peureux, F., Utting, M.: Requirements traceability in automated test generation – application to smart card software validation. In: Proceedings of ICSE 2005 Workshop on Advances in Model-Based Software Testing (A-MOST), ACM (2005)
18. Pnueli, A.: Temporal semantics of concurrent programs. In: Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science, IEEE Computer Society (1977) 46–57
19. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Workshop on Logic in Programs. Number 131 in LNCS, Springer (1981) 52–71
20. Hong, H.S., Lee, I., Sokolsky, O., Ural, H.: A temporal logic based coverage theory of test coverage and generation. In: Tools and Algorithms for the Construction and Analysis of Systems: 8th International Conference (TACAS 2002). Number 2280 in LNCS, Springer (2002) 327–339