Author(s)	Jääskeläinen, Antti
Title	Filtering test models to support incremental testing
Citation	Jääskeläinen, Antti 2010. Filtering test models to support incremental testing. Lecture Notes in Computer Science vol. 6303, 72-87.
Year	2010
DOI	http://dx.doi.org/10.1007/978-3-642-15585-7_8
Version	Post-print
URN	http://URN.fi/URN:NBN:fi:tty-201404251148
Copyright	The final publication is available at link.springer.com.

Filtering Test Models to Support Incremental Testing

Antti Jääskeläinen

Tampere University of Technology, Department of Software Systems
PO Box 553, FI-33101 Tampere, Finland
antti.m.jaaskelainen@tut.fi

Abstract. Model-based testing can be hampered by the fact that a model depicting the system as designed does not necessarily correspond to the product as it is during development. Tests generated from such a model may be impossible to execute due to unimplemented features and already known errors. This paper presents a solution in which parts of the model can be filtered out and the remainder used to generate tests for the implemented portion of the product. In this way model-based testing can be used to gradually test the implementation as it becomes available. This is particularly important in incremental testing commonly used in industry.

Keywords: Model-Based Testing, Test Modeling, Model Filtering, Model Transformation, Strong Connectivity

1 Introduction

Traditionally software test automation has focused on automating the execution of tests. A newer approach, model-based testing, allows the automation of the creation of tests by generating them from a formal model which depicts the expected functionality of the system under test (SUT). An excellent approach in theory, widespread deployment of model-based testing is nonetheless hindered by a number of practical issues.

One such issue is fitting model-based testing into the product life cycle. The error-detection capability of model-based testing is based on the correspondence between the model and the SUT; a difference between the two indicates an error in one or the other. However, testing should begin before a fully functional SUT is available, which means that this correspondence is in practice broken.

The problem first appears during the early implementation of the product. The test model can be created based on the design plans, and is likely to be ready long before all the features of the SUT have been fully implemented, since modeling is a good method of static testing. In this case, the tests generated from the model may span the whole system under development, even though the SUT only contains limited functionality. Developing and updating the model alongside the product is possible but impractical; it should be possible to model the whole system before it is fully implemented. How, then, can we use a model of the complete system to generate tests just for the current implementation?

A similar situation is encountered when the testing pays off and an error is found. Fixing the error may take some time, especially if it is particularly complicated or not very serious. Testing, of course, should be continued immediately. But how can we ensure that new generated tests do not stumble on the same, already known issue?

In these cases, the problem is that the model contains functionality that cannot be executed on the SUT, yet we need to generate actually executable tests. The magnitude of the problem depends on how the tests are generated. If the process is cheap, it may be possible to generate an overabundance of tests and discard the unfeasible ones. However, if test generation is complicated and costly, it will be necessary to ensure that as little effort as possible is wasted on unproductive tests.

This paper presents a solution based on *filtering* the test model in such a way that unimplemented or faulty functionality is effectively removed. The remainder of the model can then be used to generate tests for the implemented functionality. As new features are implemented they can be allowed into the model and test generation; as erroneous functionality is uncovered it can be filtered out until fixed. Using this method, a complete test model can be used to generate tests as soon as the product is mature enough for automatic test execution. The challenge is to ensure that the filtered model remains suitable for test generation.

The rest of the paper is structured as follows: Section 2 provides an overall presentation on our approach to model-based testing. Section 3 explains our filtering methodology in detail, and Section 4 presents a case study based on it. Finally, Section 5 concludes the paper.

2 Background

Model-based testing is a testing methodology which automates the generation of tests. This is done with the help of a *test model*, which describes the behavior desired in the tests. Depending on the approach, this may mean the behavior of the SUT or its user, or both combined.

There are two ways to execute the generated tests. In *off-line testing* the model is first used to create the test cases, which are then executed just as if they had been designed manually. In the alternate approach, *online testing*, the tests are executed as they are being generated. The latter method is especially well suited for testing nondeterministic systems, since the results of the execution can be continuously fed back into test generation, which can then adapt to the behavior of the SUT.

Our research focuses on online testing based on behavioral models. The formalism in our models is labeled state transition system (LSTS), a state machine with labeled states and transitions. LSTS is a simple formalism and other behavioral models can be easily converted into it, which allows us to create models also in other formalisms, if need be. The formal definition of LSTS is the following:

Definition 1 (LSTS).

A labeled state transition system*, abbreviated LSTS, is defined as a sextuple* $(S, \Sigma, \Delta, \hat{s}, \Pi, val)$ *where* S *is the set of* states*,* Σ *is the set of* actions *(transition labels),* $\Delta \subseteq S \times \Sigma \times S$ *is the set of* transitions*,* $\hat{s} \in S$ *is the* initial state*,* Π *is the set of* attributes *(state labels) and* $val : S \longrightarrow 2^{\Pi}$ *is the* attribute evaluation function*, whose value* $val(s)$ *is the set of attributes in effect in state* s*.*

Creating a single model to depict the whole SUT is virtually impossible for any practical system. Therefore we create several *model components*, each depicting a specific aspect of the SUT, and combine these into a test model in a process called *parallel composition*. We use a parallel composition method developed in [7], generalized from CSP (Communicating Sequential Processes) [11]. It is based on a rule set which explicitly specifies which actions are executed synchronously. The formal definition is as follows:

Definition 2 (Parallel composition $\|_R$).

$\|_R$ $(L_1, \ldots, L_n)$ *is the* parallel composition *of LSTSs* $L_1, \ldots, L_n$, $L_i = (S_i, \Sigma_i, \Delta_i, \hat{s}_i, \Pi_i, val_i)$*, according to* rules R*, such that* $\forall i, j; 1 \leq i < j \leq n : \Pi_i \cap \Pi_j = \emptyset$*. Let* Σ_R *be a set of resulting actions and* $\surd$ *a "pass" symbol such that* $\forall i; 1 \leq i \leq n : \surd \notin \Sigma_i$*. The rule set* $R \subseteq (\Sigma_1 \cup \{\surd\}) \times \cdots \times (\Sigma_n \cup \{\surd\}) \times \Sigma_R$*. Now* $\|_R$ $(L_1, \ldots, L_n) = repa((S, \Sigma, \Delta, \hat{s}, \Pi, val))$*, where*

- $S = S_1 \times \cdots \times S_n$
- $\Sigma = \Sigma_R$
- $((s_1, \ldots, s_n), a, (s'_1, \ldots, s'_n)) \in \Delta$ *if and only if there is* $(a_1, \ldots, a_n, a) \in R$ *such that for every* i $(1 \leq i \leq n)$ *either*
  - $(s_i, a_i, s'_i) \in \Delta_i$ *or*
  - $a_i = \surd$ *and* $s_i = s'_i$
- $\hat{s} = (\hat{s}_1, \ldots, \hat{s}_n)$
- $\Pi = \Pi_1 \cup \cdots \cup \Pi_n$
- $val((s_1, \ldots, s_n)) = val_1(s_1) \cup \cdots \cup val_n(s_n)$
- *repa is a function restricting LSTS to contain only the states which are reachable from the initial state* $\hat{s}$*.*

The parallel composition allows us to use a relatively small number of simple model components to create a huge test model. In practice, the test model may well be too large to calculate in its entirety, so the parallel composition is usually performed *on the fly* for the needed portion of the model. The available model components comprise a *model library* [6], from which individual components can be composed into a suitable test model.

The model components are divided into two tiers corresponding to the concepts of action words and keywords [1, 4]. *Action words* define user actions, such as those commonly used in use case definitions. Accordingly, the upper tier models based on action words, called *action machines*, describe the functionality of the SUT. Action words and action machines are independent of implementation, and can often be reused in testing other similar systems.

Keywords describe UI events, such as pressing keys or a text appearing on a display. The lower tier models, *refinement machines*, use keywords to define implementations for the action words in the action machines. Refinement machines are specific to implementation, so every different type of SUT requires its own.

The execution of a keyword returns a Boolean value, which tells whether the SUT executed the keyword successfully or not. Usually a certain value is expected, and a different result indicates an error. However, in online testing of nondeterministic systems it may be reasonable to accept either value, since the exact state of the SUT may not be known. This is modeled by adding a separate transition for successful and unsuccessful execution. The actions of such transitions are *negations* of each other. These *branching keywords* allow the implementations of action words to adapt to the state of the SUT. If the nondeterminism affects the execution of the test beyond a single action word, a similar *branching action word* is needed. Such action words can be used to direct an online test into an entirely different direction depending on the state of the SUT. Branching actions do not fit well into the linear sequences of off-line testing, though, and the unpredictability especially at the action word level makes the generation of online tests somewhat more difficult.

Tests are generated with guidance algorithms based on coverage requirements. A *coverage requirement* [8] defines the goal of the test, such as executing all actions in the model or a sequence of actions corresponding to a use case. A *guidance algorithm* is a heuristics whose task is to decide how the test will proceed. A straightforward algorithm may simply seek to fulfill the coverage requirement as quickly as possible. Others may perform additional tasks on the side, such as continuously switching between different applications in order to exercise concurrency features; yet another may be completely random.

Facilitating such diverse goals and methods places some requirements for the test model. The most important of these is that the model must be *strongly connected*, that is, all states must be reachable from all other states. A test model that is not strongly connected poses great difficulties for test generation, since the execution of any transition may render portions of the model unreachable for the remainder of the test run. Coverage requirements can no longer be combined freely, since their combination may be impossible to execute even if they are individually executable. Finally, online test generation becomes effectively impossible, because the only way to ensure that the whole test can be executed is to calculate it out entirely before beginning the execution and making potentially irreversible choices.

If strong connectivity is for some reason broken, it must be restored by limiting the model to the maximal strongly connected portion of the model containing the initial state, which we will call the *initial strong component*. Unfortunately, finding the initial strong component can be difficult if the model is too large to calculate in its entirety. In particular, strong connectivity of model components does not in itself guarantee strong connectivity in the composed test model.

Ensuring the strong connectivity and general viability of the models is in the end up to the *test modeler*, who is responsible for the creation and maintenance of the models. The *test designers*, who are responsible for the actual test runs, should be able to use the models for test generation without needing to worry about their internal structure. Such distribution of concerns relieves most of the testing personnel from the need of specialized modeling expertise [9].

3 Filtering

In this section we present our filtering method. First we go through some basic requirements for the method, and then present a solution based on those. After that, we examine implementation issues concerning the filtering process, especially regarding strong connectivity. Following is some analysis of the algorithm used in implementation, and finally an example of its use.

3.1 Basic Criteria

A method for filtering out unwanted functionality from the models should fulfill the following criteria:

1. The execution of faulty or unimplemented transitions can be prevented.
2. The model should not be restricted more than necessary.
3. The model must remain strongly connected.
4. Filtering may not require modeling expertise or familiarity with the models.
5. The manual effort involved in the process may not be excessive.
6. Filtering must be performed without modifications to the models themselves.

The first three criteria define the desired result for the filtering process. Criterion 1 is the very goal of the filtering process. Criterion 2 is likewise obviously necessary, since we want to keep testing the SUT as extensively as possible. Criterion 3 ensures that the process does not break the basic requirement placed on the test model. As a consequence, the filtering cannot be performed by just *banning* (refusing to execute) problematic transitions or actions, since such a strategy might effectively lead to deadlocks or otherwise break the strong connectivity necessary for test generation.

The next two criteria are procedural requirements. Criterion 4 requires that the filtering process can be performed with no manual involvement with the models. Ideally, the process would be carried out by test designers, who may not be familiar with the models or the formal methods involved [9]. Since the process may need to be carried out often and repeatedly, Criterion 5 states that it may not require much manual effort.

Finally, Criterion 6 is an implementation requirement. Modifying the models for filtering purposes would require extensive tool support, so that individual changes could be made and rolled back as needed, all without breaking the models. Enabling such a feature might also place additional requirements on the structure of the models.

3.2 Methodology

There are a number of potential methods by which the tester might perform the filtering of banned functionality. Most of these require additional actions in order to keep the model strongly connected, as per Criterion 3; however, with properly designed models such actions can be automated. The examined methods are:

1. Ban the execution of specific transitions of the composed test model.
2. Ban the execution of specific transitions within model components.
3. Ban the execution of specific actions.
4. Remove model components from the composition.

Actions are general labels for the events of the SUT, whereas transitions represent the SUT moving from a specific state to another through such an event; therefore, banning an individual action corresponds to banning all of the transitions labeled with it. Likewise, banning a transition from a model component may correspond to banning several transitions from the composed test model.

Method 1 fulfills all of the specified criteria except Criterion 5, where it fails spectacularly. An individual faulty transition in a model component is likely to correspond to many transitions in the test model. Even if the problem is a concurrency issue and appears only with a specific combination of applications, it is unlikely to be limited to a situation where all of the tested applications are in exactly specific states. As such, the method is thoroughly impractical.

Method 2 is more promising, since removing the faulty transition from a model component will remove all of its instances from the test model. This method is no longer minimal (Criterion 2): in case of a concurrency issue, this method may remove more functionality than is strictly necessary. However, it does not greatly limit continued testing; furthermore, a more specific method based on multiple components at once would likely require a deeper understanding of the models, violating Criterion 4. Another problem is that transitions do not have inherent identifiers, although they can be uniquely identified by their source state and action. States are only identified with numbers, whose use would at the very least require some inspection of the model components.

In practice, Method 3 works very much the same as Method 2. It may restrict the models more, but only if the model component uses the same action in multiple places, only one of which actually fails. Unlike transitions, actions are clearly labeled and test designers will work with them in any case, so they can be easily used also for this purpose.

Finally, Method 4 is also easy to use. In fact, it might well be worth implementing for other purposes such as limiting the size of the test model. However, removing whole components from the model goes against Criterion 2, since it could drastically reduce the amount of functionality available for testing. It does have one additional benefit: it is relatively easy to design the models so that the removal of a component leaves the rest of the test model strongly connected.

Of these four, Method 3, based on banning actions, appears to be the best. It does not restrict the models much more than is necessary and is quite easy to use.

It does require some additional effort in order to retain the strong connectivity of the models, though.

In contrast, Methods 1 and 2 involve serious procedural issues and in practice do not leave much more of the model available. On the other hand, Method 4 is considerably more restrictive than necessary. However, as mentioned, it may be worth implementing anyway for other reasons, in which case it can be also used to filter models where suitable.

3.3 Banning Actions

There are three implementation issues to take care of. First, we need a means to obtain a test model with individual actions removed without altering the original models, as per Method 3 and Criterion 6. Second, we must devise a method for restoring the strong connectivity of the test model (Criterion 3), since removing individual actions may break it. Third, we must take into account the branching actions, whose both branches must be retained or removed together.

The simplest way to obtain a modified test model is to create a modified copy of the rules of parallel composition such that banned actions will not show up in the test model. This method is simple to implement and limits modifications to one place. Alternatively, modified copies of the model components could be created with banned actions removed, and then composed as usual. However, such an approach would require modifications in several places, and modifying a model component is liable to be more difficult than removing rules from a list.

Ensuring the strong connectivity of the test model is more difficult. It is obviously not possible to design all models so that any actions could be removed without breaking strong connectivity. As for automation, in a general case it is not possible to determine whether a test model is strongly connected without calculating it entirely, which may be impossible due to the potential size of the model. As a solution, our filtering algorithm seeks to deduce the initial strong component from the model components and the rule set, but without calculating the parallel composition. The result is an upper bound for the initial strong component, that is, a limited portion of the original model which contains the initial strong component. The algorithm is based on the following principles:

1. an action must be banned if it labels a transition which leads away from the initial strong component of a model component
2. an action may be banned if it does not label any transition within the initial strong component of a model component
3. an action may be banned if there remain no rules which allow its execution
4. a rule may be removed if any of its component actions is banned

The first principle is the most important: leaving the initial strongly connected component of a model component cannot be allowed, since there would be no way back, and the strong connectivity of the test model would be broken. In contrast, the other three principles ban actions and remove rules which could not be executed in the test model anyway. Actions outside the initial strong components are effectively unreachable, an action without rules does not appear in the composed test model, and a rule without all of its actions can never be applied. Therefore, these three do not limit the models needlessly. They are also not useful in themselves, but may allow greater application of the first principle.

Based on these principles, we have developed Algorithm 1 and implemented it as a part of the TEMA open source toolset [10]. The lines from 1 to 11 set the initial values for the data structures, as well as marking for handling the initially banned actions and removed rules. The loop on line 12 additionally marks for handling those actions for which there are no rules. The three main parts of the algorithm are within the loop on line 16. First, the loop on line 18 handles banned actions, removing any rule which requires them. Second, the loop on line 24 handles rules in a similar way, banning all actions for which there are no rules left. Third, the loop on line 32 calculates the initial strong components of the model components and marks for handling those actions which lead outside the component or cannot be reached within it. These three are repeated until no more actions can be banned or rules removed. The calculation of the strong components, which can be performed for example by Tarjan's algorithm [13], is the most time-consuming part of the algorithm. It is therefore only performed when no other method for progress is available.

The algorithm returns both a set of removed rules and one of banned actions; either can be used to perform the actual filtering. The list of banned actions is also useful to the modeler, since it can be used to estimate the effects of filtering. This is important because the algorithm does not necessarily yield the exact initial strong component but only an upper bound for it. The rest will be up to the modeler, who should design the models so that the bound is in fact exact, and there is no way out of the initial strong component.

The nature of the algorithm makes it easy to define not only an initial set of banned actions, but also one of removed rules. This may be occasionally useful, for example to remove some kinds of actions across the model components.

Specific model semantics may require some changes or additions to the basic algorithm. Branching actions are such a case: if one branch gets banned, the other one must, too. To take this into account, we modify the algorithm such that every time an action is marked to be handled, we check for other branches and mark them also. It might also be useful to allow the modeler to define similar dependencies on a case-by-case basis, where strong connectivity demands it; we have yet to implement such a method, however.

3.4 Analysis

Following is a brief analysis of the time requirements of Algorithm 1. For an arbitrary model component $m \in M$, we will mark $m = (S_m, \Sigma_m, \Delta_m, \hat{s}_m, \Pi_m, val_m)$. All set operations used in the algorithm (addition and removal of elements, check for membership or emptiness) can be performed in amortized constant time.

The handling of each rule requires $O(|M|)$ time: it may get marked for handling by each action it refers to, and may have to mark for handling each of

Algorithm 1 The filtering algorithm for the set of model components M composed with the rules R, with the rules $remove \in R$ initially removed and the actions $ban(m) \in \Sigma_m$ of model components $m \in M$ initially banned.

```
$banned\_actions, unhandled\_actions, removed\_rules := \emptyset$
$unhandled\_rules := remove$
$changed\_models := M$
**for all** model components $m \in M$ **do**
    **for all** actions $a \in ban(m)$ **do**
        add $(m, a)$ to $unhandled\_actions$
    **for all** actions $a$ of $m$ **do**
        $remaining\_rules(m, a) := \emptyset$
**for all** rules $r \in R$ **do**
    **for all** actions $a$ of model components $m$ in $r$ **do**
        add $r$ to $remaining\_rules(m, a)$
**for all** model components $m \in M$ **do**
    **for all** actions $a$ of $m$ **do**
        **if** $remaining\_rules(m, a) = \emptyset$ **then**
            add $(m, a)$ to $unhandled\_actions$
**while** $unhandled\_actions \neq \emptyset$ or $unhandled\_rules \neq \emptyset$ **do**
    **while** $unhandled\_actions \neq \emptyset$ or $unhandled\_rules \neq \emptyset$ **do**
        **for all** model-action pairs $(m, a) \in unhandled\_actions$ **do**
            **for all** rules $r \in remaining\_rules(m, a)$ **do**
                **if** $r \notin removed\_rules$ **then**
                    add $r$ to $unhandled\_rules$
            add $(m, a)$ to $banned\_actions$
        $unhandled\_actions := \emptyset$
        **for all** rules $r \in unhandled\_rules$ **do**
            **for all** actions $a$ of model components $m$ in $r$ **do**
                remove $r$ from $remaining\_rules(m, a)$
                **if** $remaining\_rules(m, a) = \emptyset$ and $(m, a) \notin banned\_actions$ **then**
                    add $(m, a)$ to $unhandled\_actions$
                    add $m$ to $changed\_models$
            add $r$ to $removed\_rules$
        $unhandled\_rules := \emptyset$
    **while** $changed\_models \neq \emptyset$ and $unhandled\_actions = \emptyset$ **do**
        $m :=$ any element from $changed\_models$
        remove $m$ from $changed\_models$
        $reachables := \emptyset$
        $isc :=$ the initial strong component of $m$ with banned actions removed
        **for all** transitions $(s, a, s')$ of $m$ **do**
            **if** $s$ within $isc$ **then**
                add $a$ to $reachables$
                **if** $s'$ not within $isc$ and $(m, a) \notin banned\_actions$ **then**
                    add $(m, a)$ to $unhandled\_actions$
                    add $m$ to $changed\_models$
        **for all** actions $a$ of $m$ **do**
            **if** $a \notin reachables$ and $(m, a) \notin banned\_actions$ **then**
                add $(m, a)$ to $unhandled\_actions$
                add $m$ to $changed\_models$
**return** $removed\_rules, banned\_actions$
```

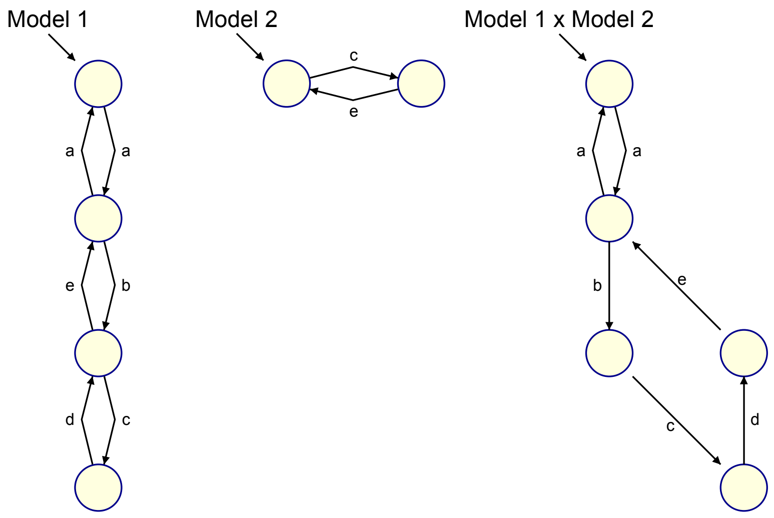

Fig. 1. Two example model components and their composition with the rules $R = \{(a, \surd, a), (b, \surd, b), (c, c, c), (d, \surd, d), (e, e, e)\}$.

those actions. For all rules, this gives $O(|R||M|)$. In addition to this, the handling of each action takes only constant time, yielding $O(\sum_{m \in M} |\Sigma_m|)$. Calculating the strong components of a single model $m \in M$ with Tarjan's algorithm takes $\Theta(|S_m| + |\Delta_m|)$ time. However, since we are only interested in the initial strong component, effectively $|S_m| \leq |\Delta_m| + 1$, resulting in $\Theta(|\Delta_m|)$. The subsequent handling requires $\Theta(|\Delta_m| + |\Sigma_m|) = \Theta(max(|\Delta_m|, |\Sigma_m|))$. The calculation is carried out for each model only after new actions have been banned; since all unreachable actions get banned on the first (compulsory) time, the calculation will be performed at most $min(|\Sigma_m|, |\Delta_m|) + 1$ times. The result is $O(\sum_{m \in M} min(|\Sigma_m|, |\Delta_m|) max(|\Delta_m|, |\Sigma_m|)) = O(\sum_{m \in M} |\Sigma_m||\Delta_m|)$.

Putting the above figures together, we get $O(|R||M| + \sum_{m \in M} |\Sigma_m||\Delta_m|)$. This means linear dependence on the number of rules times the size of a single rule, plus quadratic dependence on what is essentially the sizes of the model components. The first term is quite reasonable, since the same time is required to simply write out the rules. The second term, while not insignificant, is still perfectly manageable if individual model components are kept small enough.

3.5 Example

We will now present an example of Algorithm 1 with the models in Figure 1, combined with the rules $R = \{(a, \surd, a), (b, \surd, b), (c, c, c), (d, \surd, d), (e, e, e)\}$. Let us assume that the implementation of action d of Model 1 is faulty and initially ban $(1, d)$.

Since the action $(1, d)$ is banned, we remove the rule $(d, \surd, d)$ which refers to it. After that, we must calculate strong connectivity; we shall do it for Model 1

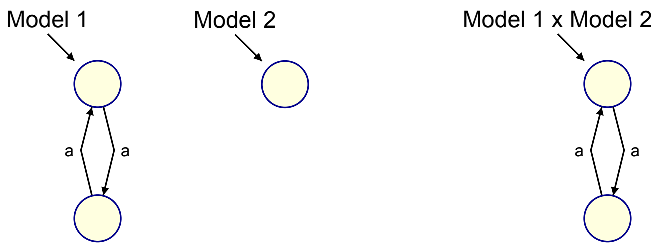

Fig. 2. Filtered versions of the example model components and their composition with the filtered rules $R = \{(a, \surd, a)\}$.

(calculating the strong connectivity for Model 2 would not yield anything new anyway). We notice that in Model 1 the action c leads out of the initial strong component and ban $(1, c)$. Consequently, we also remove (c, c, c) and then, because there are no longer any rules for it, $(2, c)$.

Again we must calculate strong connectivity. This time, we do not learn anything from calculating it for Model 1, but in Model 2 we notice that $(2, e)$ is unreachable and ban it. Following that, we remove (e, e, e) and ban $(1, e)$. We note that now the action b breaks the strong connectivity of Model 1, and ban $(1, b)$ and remove $(b, \surd, b)$. Finally, Model 2 has changed since our last connectivity calculation for it, so we perform one, but learn nothing new. At this point the algorithm returns the results and terminates.

In the end, we have banned the actions b, c, d and e from Model 1; banned the actions c and e from Model 2; and removed the rules $(b, \surd, b)$, (c, c, c), $(d, \surd, d)$ and (e, e, e). All that is left of the model components is a two-a loop in Model 1, which is also exactly what will show up in the test model composed with the single remaining rule $(a, \surd, a)$, as seen in Figure 2. Looking at the original composed model in Figure 1, it is easy to see that this is what should happen with the action d banned.

3.6 Other Composition Methods

If the algorithm is to be used with a different method of parallel composition, it will be necessary to create a rule set that implements corresponding functionality. For example, the basic parallel composition where actions of the same name are always executed synchronously would correspond to the rules

$$\begin{aligned} R = \{&(\sigma_1, \dots, \sigma_n, \sigma_R) \in (\Sigma_1 \cup \{\surd\}) \times \dots \times (\Sigma_n \cup \{\surd\}) \times (\Sigma_1 \cup \dots \cup \Sigma_n) \mid \\ &\forall i; 1 \leq i \leq n : (\sigma_R \in \Sigma_i \rightarrow \sigma_i = \sigma_R) \land (\sigma_R \notin \Sigma_i \rightarrow \sigma_i = \surd)\} \end{aligned}$$

Although the rule set is needed for the execution of the algorithm, it is not necessary to actually implement rule-based parallel composition. The list of banned actions the algorithm returns can be used to perform filtering within the model components, and these can then be combined with the original method of composition.

4 Case Study

As a case study, we will examine the process of modifying models from an existing model library to conform to the requirements of filtering. The purpose is to ensure that test models composed from the library can be relied on to remain strongly connected when arbitrary actions are filtered out; afterward, filtering can be performed automatically. First, we will present the model library and how its model components might in practice be filtered. We will then examine the actual modifications made to the models of one application in the library, and finally analyze the results.

4.1 Setup

The model library we will examine has been designed for the testing of smartphone applications [5]. The latest version contains models for eight applications such as Contacts and Messaging, over four different phone models, on different platforms such as S60 and Android. The model components in the library have been designed to yield a usable test model even if only some of them are included in the composition, as long as specified dependencies are met. However, they have not been designed to withstand the arbitrary removal of actions gracefully.

In this case study we will focus on the models of the Contacts application. It consists of six action machines and a corresponding number of refinement machines, and has about 330 states altogether. As such it is one of the smaller applications in the library, and simple enough to be a comprehensible example.

When examining the effects of filtering, we can safely limit ourselves to banning action words in the action machines, since they represent the (potentially unavailable) functionality of the SUT. The task is performed by banning action words one at a time and examining the results with the help of the filtering algorithm. From the results we can determine whether the composed test model would remain strongly connected or not.

4.2 Modifications

An initial execution of the algorithm with no actions banned yields a list of a few unimplemented actions; these appear in the action machines but have no implementation. Such actions would not appear in the test model anyway, so they can be safely banned. We then proceed to banning individual action words, and find two problematic situations.

The first problem we encounter is in the model component depicting the functionality of the list of contacts (Figure 3). The only action word in the model, *awVerifyContactsExist*, is a branching action word used to find out whether there are any contacts in the application (the negative branch is prefixed with a '$\sim$'). This action can only be executed if we are unsure of the current situation regarding contacts; the preceding synchronization actions check from other model components whether we know anything about the existence of contacts.

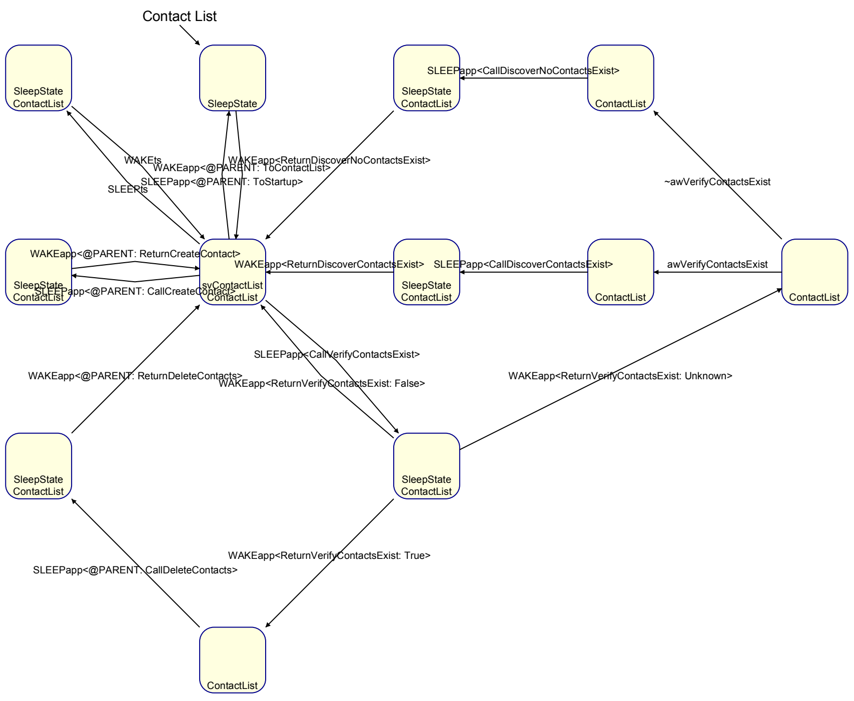
Contact List
SleepState ContactList
SleepState
SleepState ContactList
ContactList
SLEEPapp<CallDiscoverNoContactsExist>
WAKEts
SLEEPts
WAKEapp<@PARENT: ToContactList>
SLEEPapp<@PARENT: ToStartup>
WAKEapp<ReturnDiscoverNoContactsExist>
~awVerifyContactsExist
WAKEapp<@PARENT: ReturnCreateContact>
SLEEPapp<@PARENT: CallCreateContact>
SleepState ContactList
svContactList ContactList
WAKEapp<ReturnDiscoverContactsExist>
SleepState ContactList
SLEEPapp<CallDiscoverContactsExist>
ContactList
awVerifyContactsExist
ContactList
SLEEPapp<CallVerifyContactsExist>
WAKEapp<@PARENT: ReturnDeleteContacts>
WAKEapp<ReturnVerifyContactsExist: False>
WAKEapp<ReturnVerifyContactsExist: Unknown>
SleepState ContactList
SleepState ContactList
WAKEapp<ReturnVerifyContactsExist: True>
SLEEPapp<@PARENT: CallDeleteContacts>
ContactList

Fig. 3. The Contacts List action machine, with the action word *awVerifyContactsExist* on the right.

The filter algorithm quite intuitively suggests that if the action word is banned, the action *WAKEapp<ReturnVerifyContactsExist: Unknown>* should also be banned to preserve strong connectivity. However, that would actually cause a deadlock elsewhere in situations where the existence of contacts really is unknown. The solution here is to add a transition with a new comment action from the state on the right between the synchronization and the action word back to the central state on the middle left. A comment action can be executed with no effect to the other model components or the SUT, allowing us to bypass the verification of contacts' existence. Now the synchronizing action no longer needs to be removed with the action word, and strong connectivity is preserved.

The second problem spot is also related to the way the models keep track of the number of contacts. The existence of contacts is abstracted into three categories: contacts exist, contacts do not exist, and unknown, with unknown used as the initial value. The problem shows up in the model component responsible for the deletion of contacts (Figure 4), if we ban one of the actions *awToggleContact*, *awAttemptDelete* or *awDelete*.

The immediate result of the ban is that contacts can no longer be removed individually (or at all for *awDelete*). However, the individual removal of contacts

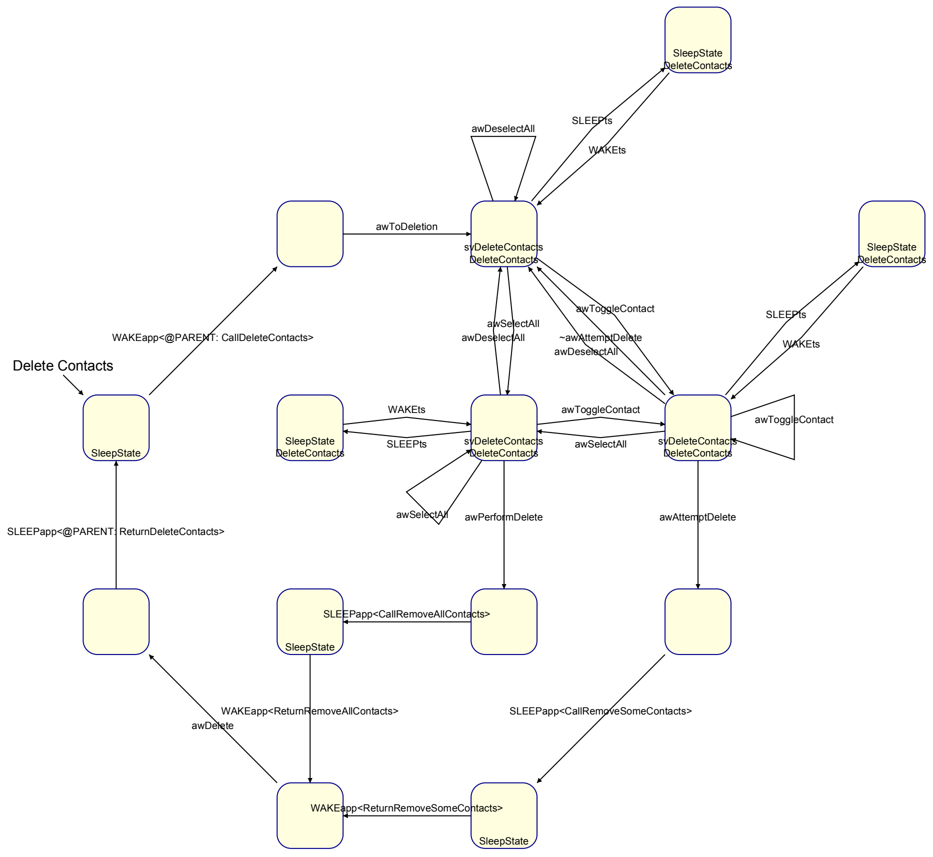

Fig. 4. The Delete Contacts action machine, with the action words *awToggleContact* and *awAttemptDelete* at the right side of the octagon, and *awDelete* at the bottom left. *awAttemptDelete* fails if no contacts are selected.

is the only way that the existence of contacts, once known, can become unknown again. This means that their existence cannot ever be allowed to become known, which results in banning every action related to their creation and handling. The test model becomes next to useless, though is does remain strongly connected. Despite the apparent complexity of the problem, the solution is simple: modify the models so that the knowledge of the existence of contacts can be 'forgotten', moving us back into the unknown state.

4.3 Results

All in all, the Contact models withstood the banning of action words fairly well. The first described problem is likely typical, with complex synchronizations between the model components resulting in a deadlock whose existence the filtering algorithm cannot deduce. The second problem shows that broken strong connectivity is not the only potential issue; one should also consider whether connectivity could be preserved with lesser limitations.

The filtering algorithm was very useful in finding the problematic situations in the models. While the first problem would have been easy enough to spot in manual inspection, the second was more obscure and might have been easily missed. Using the algorithm to calculate the effects of removing actions was also much faster than manual examination would have been.

Making the necessary modifications to the models clearly requires some modeling expertise. This is not a serious issue, since they would usually be made by the original modeler, as part of the normal modeling process. In this case the whole modification process took less than an hour, and was performed manually apart from using the filtering algorithm. Thus, there should not be any significant increase in the modeling effort.

5 Discussion

Using model-based testing in the early phases of product implementation can be difficult, because the product does not yet correspond to the model depicting the entire system. The problem can be solved by altering the model so that unimplemented or faulty functionality is removed and no tests are generated for it. This way the model can be matched to the product throughout its implementation.

Model transformations [2] can be used to modify the test models as needed; their use to keep the test models up to date during development is described in [12]. The use of parallel composition to limit the model to specific scenarios is mentioned in [3, 14], although no mention is made of ensuring the viability of the resulting models. All in all, there does not appear to be much previous work on restricting the functionality of test models and the consequences thereof.

The basic method presented in our paper is very simple, based on banning the actions corresponding to unexecutable functionality in the models or removing the rules acting on them in the parallel composition. The greatest challenge is ensuring that the model remains conducive to test generation; specifically that it remains strongly connected. The algorithm presented in the paper seeks to estimate the initial strong component of the model as well as possible without actually calculating the composed test model. The rest is left up to the modeler.

Our case study showed that modifying existing models to withstand filtering without losing strong connectivity is feasible; by extension, so is designing models to match the same requirement from the first. The filtering algorithm proved very useful in the task, since it can be used to show the effects of banning specific actions and thus reveal problematic structures in the models.

The filtering algorithm takes advantage of the explicit set of synchronization rules used by our method of parallel composition. It can also be used with other parallel composition methods, if a suitable rule set is created to describe the synchronizations. The practical issues related to this are left for future work.

Likewise for the future are left the methods for filtering non-behavioral models and test data.

Acknowledgements The author wishes to thank Mika Katara, Shahar Maoz and Heikki Virtanen for their comments. Funding from Tekes, Nokia, Ixonos, Symbio, Cybercom Plenware, F-Secure, Qentinel, Prove Expertise, as well as the Academy of Finland (grant number 121012), is gratefully acknowledged.

References

1. Buwalda, H.: Action figures. STQE Magazine, March/April 2003
2. Czarnecki, K., Helsen, S.: Classification of model transformation approaches. In: OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture. pp. 324–339. Springer-Verlag (2003)
3. Ernits, J., Roo, R., Jacky, J., Veanes, M.: Model-based testing of web applications using NModel. In: TestCom/FATES. pp. 211–216. Springer-Verlag (2009)
4. Fewster, M., Graham, D.: Software Test Automation: Effective use of test execution tools. Addison–Wesley (1999)
5. Jääskeläinen, A., Katara, M., Kervinen, A., Maunumaa, M., Pääkkönen, T., Takala, T., Virtanen, H.: Automatic GUI test generation for smart phone applications - an evaluation. In: Proc. of the Software Engineering in Practice track of the 31st International Conference on Software Engineering (ICSE 2009). pp. 112–122, companion volume. IEEE Computer Society (2009)
6. Jääskeläinen, A., Kervinen, A., Katara, M.: Creating a test model library for GUI testing of smartphone applications. In: Proc. 8th International Conference on Quality Software (QSIC 2008) (short paper). pp. 276–282. IEEE Computer Society (Aug 2008)
7. Karsisto, K.: A new parallel composition operator for verification tools. Doctoral dissertation, Tampere University of Technology (number 420 in publications) (2003)
8. Katara, M., Kervinen, A.: Making model-based testing more agile: a use case driven approach. In: Proc. Haifa Verification Conference 2006. pp. 219–234. No. 4383 in LNCS, Springer (2007)
9. Katara, M., Kervinen, A., Maunumaa, M., Pääkkönen, T., Satama, M.: Towards deploying model-based testing with a domain-specific modeling approach. In: Proc. TAIC PART – Testing: Academic & Industrial Conference 2006. pp. 81–89. IEEE CS (Aug 2006)
10. Practise research group: TEMA project home page. Available at `http://practise.cs.tut.fi/project.php?project=tema`. Cited Apr. 2010.
11. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice Hall (1998)
12. Rumpe, B.: Model-based testing of object-oriented systems. In: Formal Methods for Components and Objects, International Symposium, FMCO 2002, Leiden. LNCS 2852. pp. 380–402. Springer-Verlag (2003)
13. Tarjan, R.: Depth-first search and linear graph algorithms. SIAM Journal on Computing 1(2), 146–160 (1972)
14. Veanes, M., Schulte, W.: Protocol modeling with model program composition. In: Proceedings of the 28th IFIP WG 6.1 international conference on Formal Techniques for Networked and Distributed Systems. pp. 324–339. Springer-Verlag (2008)