To appear in Proc. Haifa Verification Conference 2008, LNCS, Springer.
© Springer-Verlag.

Synthesizing Test Models from Test Cases

Antti Jääskeläinen, Antti Kervinen, Mika Katara, Antti Valmari, and Heikki Virtanen

Tampere University of Technology
Department of Software Systems
P.O.Box 553, FI-33101 Tampere, FINLAND
{antti.m.jaaskelainen,firstname.lastname}@tut.fi

Abstract. In this paper we describe a methodology for synthesizing test models from test cases. The context of our approach is model-based graphical user interface (GUI) testing of smartphone applications. To facilitate the deployment of model-based testing practices, existing assets in test automation should be utilized. While companies are interested in the benefits of new approaches, they may have already invested heavily in conventional test suites. The approach presented in this paper enables using such suites for creating complex test models that should have better defect detection capability. The synthesis is illustrated with examples from two small case studies conducted using real test cases from industry. Our approach is semi-automatic requiring user interaction. We also outline planned tool support to enable efficient synthesis process.

1 Introduction

Model-based software testing [1] has several obvious advantages over conventional test suite testing where test cases are crafted manually. For instance, on-line tests generated from state machines can reach significantly higher coverage in testing non-deterministic systems under test (SUTs) than linear and static test suites. Moreover, maintenance of large test suites is more difficult when changes occur in the SUT. Frequent changes are common especially in graphical user interface (GUI) testing that is typically used to check the functionality of the SUT from the perspective of the end users before a release is made.

The problems with conventional test automation approaches have resulted in many bad experiences, and manual testing is still widely considered as the primary quality assurance method at the system and acceptance level testing of GUI-intensive software [2]. While unit and integration level test automation can significantly improve code quality and enable efficient refactoring, system level test automation entails much more challenges. This is due to the *domain-specific nature* of system level testing; at the unit and integration levels all SUTs seem more or less similar, depending on the programming language used; the same white-box testing and static analysis techniques work across different domains. At the system level, however, the context comes into play: testing a banking system can be quite different from testing a set-top box.

The deployment of model-based system testing has been hampered in many contexts in spite of its many benefits [3, 4]. In our earlier work, we have developed a domain-specific solution to the GUI testing of S60 [5] smartphone applications that should be

easier to deploy than more generic methodologies [6, 7]. The approach consists of a domain-specific modeling language based on LSTSs (Labeled State Transition Systems) augmented with S60 specific restrictions, a model-library containing test models for the basic smartphone applications such as calendar, contacts, camera, and messaging, and tools for on-line test generation. In on-line testing, the idea is to generate tests while they are executed, thus testing can be seen as a game between the test automation system and the SUT [8].

In the course of developing our approach we have identified another problem in deployment: companies may have invested huge sums of money to craft test suites and thus can be unwilling to invest to the development of test models replacing the former way of working. Thus, in order to facilitate the deployment of our approach, we have developed a semi-automatic method for synthesizing test models from test cases. This enables utilizing the existing assets when moving from test suite testing to model-based one. The method is domain-specific to enable a higher level of automation in the synthesis and promote the usefulness of the resulting models. However, a similar method could presumably be developed for some other domain, using similar principles.

In this paper we describe the method and the case studies we have conducted. In addition, since model synthesis is quite different from the traditional way of creating models, and we compare the synthesized model to a one crafted by hand using a top-down approach [9]. A tool support for the synthesis is also outlined; its implementation will be future work. The remainder the paper is structured as follows: Section 2 describes the context of our contributions, i.e., model-based GUI testing of mobile applications. Then, we move on to present our approach for model synthesis in Section 3. Sections 4 and 5 present the case studies and discuss the results and the future work.

2 Model-Based GUI Testing of Mobile Software

Action words and keywords [10, 11] are commonly used concepts in software test automation, especially in GUI testing. The basic idea is to separate different concerns: *what* are the important actions to be tested and *how* they are implemented. Action words are high level descriptions of functionality; in the smartphone context there can be different action words for opening the messaging application, taking a photo with the camera, or adding a new contact, for instance. Keywords, on the other hand, specify the exact sequence of events that are needed to implement the functionality described by an action word. In S60 GUI, for instance, there can be multiple ways of opening a messaging application (short cut, menu, some other application). Each of the different ways can be encoded as a separate sequence of key strokes that accomplish the action. Furthermore, to receive input from the SUT, some keywords can be dedicated to verifying that a given text string is found on the display, for instance.

The main benefit of action words and keywords is in enabling non-technical testers to design action word level tests without deep knowledge of the underlying keyword implementations. Moreover, they ease the tedious maintenance tasks often hindering the use of GUI test automation; in many cases minor GUI changes can be restricted to the keyword level. Action words and keywords can be used in conventional approaches so that the keywords are implemented as a library of functions, one function for each

keyword. Action words are then specified using spread sheets, for instance, that list the sequences of keywords needed to implement the corresponding action word. Finally, test cases can be encoded as sequences of action words using spread sheets as in the previous step.

However, linear and static tests are limited in their ability to find new defects. Thus, the true power of the action words and keywords is realized when combined with automatic test generation based on behavioral models. For this purpose, we have chosen to use Labeled State Transition Systems (LSTSs) [12] for test modeling. LSTS is an extension of the more common Labeled Transition System (LTS) formalism where labels have been added to states as well as transitions. Action words and keywords are used as transition labels in the models. The formal definition for LSTS is as follows:

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*.*

Notation of internal transitions makes no sense in test modeling, because our behavioral models have to be strictly deterministic for test generation. Our definition differs from the original one in that respect.

Actions can be divided into three categories according to how they deal with the SUT: *input*, *output* and *setup actions*. Input actions correspond to user input, and output actions get information from the SUT. Setup actions affect the SUT just as input actions, but in ways not accessible to an ordinary user. Setup actions might, for example, directly create or remove files in memory or alter internal settings. Action words often combine aspects of more than one category, whereas keywords usually fall neatly into one or another.

To enable modular and compositional test modeling, *parallel composition* is used for combining test model components. The parallel composition of LSTSs [12] is based on a rule set explicitly defining which actions are executed synchronously. An action of the composed LSTS can be executed only if the corresponding actions can be executed in each component LSTS, or if the component LSTS is indifferent to its execution. The following definition is slightly modified in two respects; internal transitions are not needed and handling of state propositions is made more straightforward:

Definition 2 (Parallel composition $\|_R$). $\|_R\ (L_1,\dots,L_n)$ *is the* parallel composition *of LSTSs* $L_1,\dots,L_n$*,* $L_i=(S_i,\Sigma_i,\Delta_i,\hat{s}_i,\Pi_i,val_i)$*, according to* rules R*, with* $\forall i,j; 1\le i< j\le n : \Pi_i\cap\Pi_j=\emptyset$*. Let* Σ_R *be a set of resulting actions and* $\surd$ *a "pass" symbol such that* $\forall i; 1\le i\le n : \surd\notin\Sigma_i$*. The rule set* $R\subseteq(\Sigma_1\cup\{\surd\})\times\cdots\times(\Sigma_n\cup\{\surd\})\times\Sigma_R$*. Now* $\|_R\ (L_1,\dots,L_n)=repa((S,\Sigma,\Delta,\hat{s},\Pi,val))$*, where*

- $S=S_1\times\cdots\times S_n$
- $\Sigma=\Sigma_R$
- $((s_1,\dots,s_n),a,(s'_1,\dots,s'_n))\in\Delta$ *if and only if there is* $(a_1,\dots,a_n,a)\in R$ *such that for every* i $(1\le i\le n)$ *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 function restricting LSTS to contain only the states which are reachable from the initial state* $\hat{s}$.

Parallel composition offers tools for implementing rudimentary variables, which the basic LSTS formalism lacks. A variable can be created as a single component model, whose states correspond to different values. The actions in such a *variable model* are synchronized to those of the other component models so that different values allow different actions. These synchronized actions can be used to test the value of the variable or to change it. The idea of using compositional test modeling and separate variable components is motivated by existing tools, that proof the concept [13].

To hide the complexity inherent in test models and test generation algorithms, and so to facilitate the deployment of our model-based testing methodology, we have introduced a web based testing service [7]. The idea is that test service users can order tests using a simple web interface specifying the desired coverage requirements. The coverage requirements are then used for driving on-line test generation based on an extensive model library containing test models for basic S60 applications [9].

We believe that such a service can greatly ease the adoption of model-based testing in smartphone application testing. However, companies have existing assets in conventional test suites, and they might prefer to utilize them when migrating from traditional test suite based automation to a model-based one. This led us to research an approach for synthesizing test models from test cases.

3 Synthesis of Test Models

The synthesis process we have developed allows the creation of a single test model from a number of test cases. The cases must be strictly linear to begin with; they should also be specific in detail. The resulting model will have the same level of abstraction (action word/keyword) as the original cases. Test cases which verify the state of the SUT often may be easier to handle, but the process is designed to also work with few or no verifications.

The process has five distinct phases. In the first phase the relevant actions are listed and parameterized. The second phase consists of creating variables to hold some of the state information of the SUT. The third phase takes care of the initialization sequence of the SUT. In the fourth phase recurring states within the test cases are marked and labeled. Finally, the fifth phase sees the test cases merged together with the variables and the initialization to form a new test model.

Although the phases are presented consecutively, their order is not fixed. Only the merging phase is dependent on the others and must therefore be performed last. The others may be performed in any order, and it may even be a good idea to consider them side by side. Throughout the process description we will present a running example, starting with the three imaginary action word level test cases in Figure 1. In the first the phone

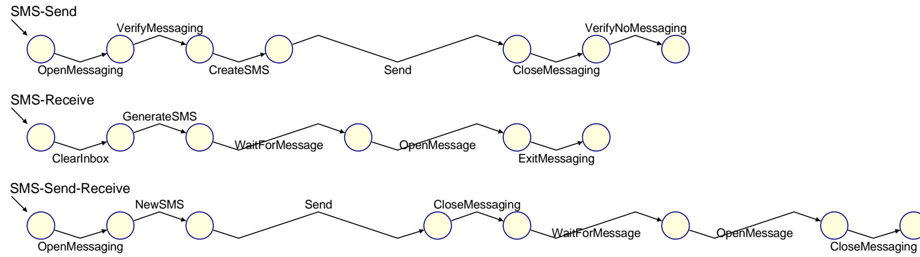

Fig. 1. The three initial example test cases.

sends an SMS to itself, in the second it receives and opens an automatically generated SMS, and in the third it first sends an SMS and then receives it. Note that the actions *CreateSMS* and *GenerateSMS* perform the same task, as do the actions *CloseMessaging* and *ExitMessaging*. They are used to demonstrate the effects of different actions sequences corresponding to the same functionality.

3.1 Action Definition

The first thing to do is to list all the actions used within the source test cases. Possible parameters should not be included. Once listed, each action is assigned two values: weight and idempotence status.

An action's weight represents its situational specificity. An action with a high weight is one whose execution with a certain parameter is likely to lead the SUT into the same state every time. This may be either because the action is only executable in very few states or because it resets parts of the SUT. An action with a low weight, on the other hand, is one which can be executed in many different situations and whose effects depend on the current situation. Weights are used in the merging of test cases. If identical action sequences taken from different test cases or different parts of the same test case have a high combined weight, it is likely that the sequences are related to the same functionality of the SUT. If this is the case, the two test cases may be merged at the points after the sequences, giving them two different ways to proceed from that point. The comparison is made with sequences instead of single actions because a long series of actions is likely to be far more situationally specific than any of its actions individually.

Actions may be marked as idempotent. The execution of an idempotent action leaves the SUT in the state it had before the execution. Most idempotent actions are used to get information out of the SUT. An idempotent action can be discarded from a test case without breaking it, although the testing value of the case may drop.

Finding the right weights is not an exact process. Action words should generally be given high weights, whereas keywords' weights vary case by case. In our running example all actions are action words. This means they have a high situational specificity, and we can give all of them maximal weights. VerifyNoMessaging and VerifyMessaging are idempotent, the rest are not.

Following are some examples with keywords: A keyword for resetting the SUT has a very high weight, since by default it always leaves the SUT in the same state. It is clearly not idempotent. A keyword which verifies that a given text is visible on the screen is idempotent and has a relatively high weight, since the same text does not very often occur in different situations. A keyword indicating that nothing should be done for a period of time has minimal weight, since waiting is always possible. It is not idempotent, because it is generally used in situations where the state of the SUT is expected to change during the wait.

3.2 Variable Definition and Integration

Embedding a part of the state of the SUT into variables is an important part of the synthesizing process. Without separate variables, the states of the test cases may contain so much information that they can never be merged together. The first, most difficult task is to identify the variables to be created. As a general rule, those properties of the SUT which are independent of the current screen of the SUT yet affect execution should be moved to variables. Having too few variables reduces the number of potential merge points and thereby limits the functionality of the final model. Too many variables mean more work in creating them and may increase the size of the final model, but should not reduce its quality.

After the variables have been determined, each is given a number of possible values. The number of values should be kept as small as possible, because they can cause exponential growth in the final model. Once the values have been chosen, each may be given one or more setup actions as *assignment actions*. In the final model, the execution of the assignment action will automatically set the variable into the designated value. A single action may act as an assignment action for multiple values, as long as they do not belong to the same variable. Finally, for each variable one of its values may be chosen as the initial value. The initial value should either have an assignment action or be otherwise guaranteed when testing begins. A variable may be left uninitialized, but then no action based on it can be taken until it has been given a value during a test run, and the size of the resulting model is also somewhat increased.

Once the variable definitions are ready, variable models are created for them. For this purpose we have made a simple Python script which reads in the variable definitions in CSV (Comma Separated Values) format and automatically produces an LSTS for each variable. The script also creates a *variable initialization model* which can set the variables to specific values before a test run by using the assignment actions.

The ready variables must be integrated into the test cases. This is performed by adding preconditions and postconditions to the actions in the test cases. Preconditions specify the values of the variables necessary for the successful execution of the action. Postconditions, conversely, define the changes of values caused by the execution of the action. Assignment actions do not require explicit postconditions, but are synchronized directly into appropriate variables. For optimal result, pre- and postconditions should be placed right around the relevant action, not around a whole action sequence.

In our running example, we create a single variable to record whether there is a message on its way to the phone, so that we will be free to merge the test cases at the main screen, regardless of whether messages have been sent or not. We use GenerateSMS

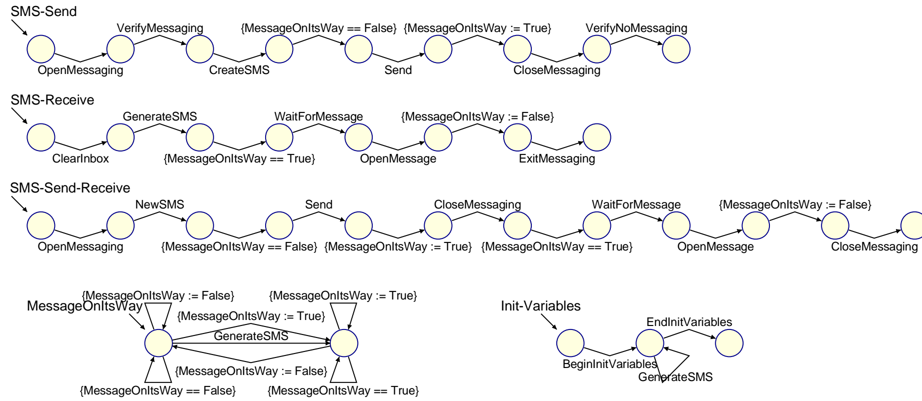

Fig. 2. The example test cases with pre- and postconditions added.

as an assignment action for the value True and pick False as the initial value, which should be safe for a new test run. Figure 2 shows the variable model and the variable initialization model, and above them the test cases with pre- and postconditions marked with braces.

3.3 Initialization Sequence Definition

In order to automatically set the SUT into its initial state before a test run, an initialization sequence is defined. The sequence contains those setup actions which should always be executed before a test run. They could, for example, reset the SUT, disable features that might interfere with testing, and create suitable data. Variable initialization should not be included here. As a rule, all setup actions should be within the initialization sequence or act as an assignment action for a variable. If a setup action belongs to neither group, more variables might be needed.

The rest of the initialization phase could be performed automatically with the information from the earlier phases, although we do not currently have tools for it. The initialization sequence is made into a *general initialization model*. All non-idempotent setup actions are removed from the beginnings of the test cases (by now they are all in the general initialization model or the variable initialization model), and synchronization is added to connect them into the initialization models.

The changes made into the test cases in the example are very minor, as Figure 3 shows. The only setup action is ClearInbox, which has been moved into a model of its own.

3.4 State Label Definition and Assignment

The existence of the variables allows the test cases to be merged with relative freedom, but there is no guarantee that suitable merging points can be automatically identified.

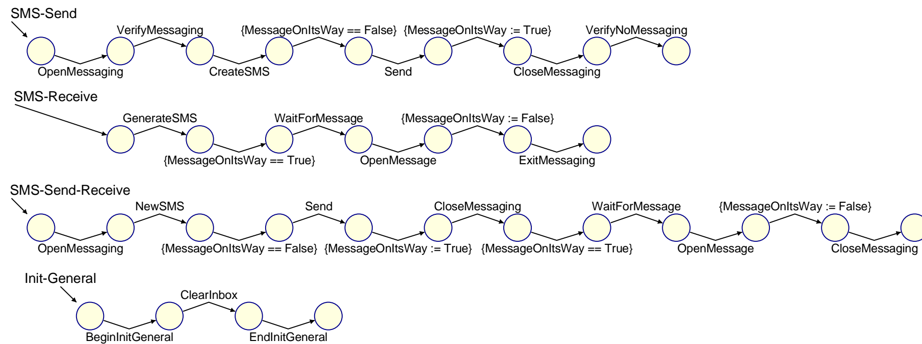

Fig. 3. The example test cases with setup actions separated.

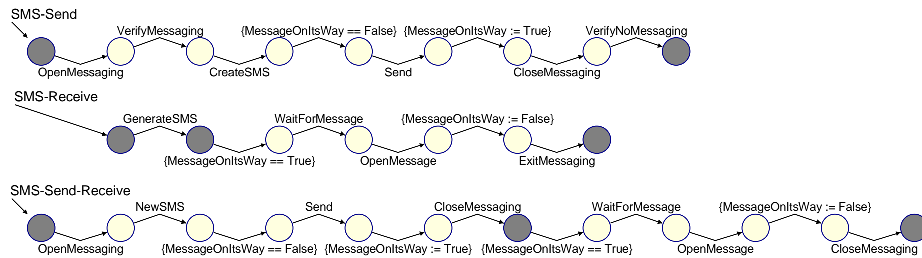

Fig. 4. The example test cases with filled states marking the main screen.

For this purpose *state labels* are added into the test cases. The important states of the SUT are identified and a name is given to each. Especially important are the starting and ending states of the test cases (ideally the same state); the basic states of other major SUT screens visited during the test cases are also good choices. Properties included in variables should be ignored.

Once the important states have been selected, state labels with suitable parameters are placed into test cases at every point in which the SUT is in a chosen state. The state labels can be handled as LSTS attributes; alternatively they can be interpreted as idempotent actions with maximal weights. Either way, merges will always be attempted at their points of execution. They can be easily removed from the final model so that they do not interfere with its execution.

In our example, we decide that the only noteworthy state is the main screen of the phone and label it, as shown in Figure 4. The states in question have been filled.

3.5 Merging of the Component Models

Now that the test cases have been prepared we can perform the actual merging. This is done with the merger program, which looks for identical sequences of sufficient weight

Fig. 5. The model merged from example test cases.

within the test cases and suggests merging their destination states. The program may also offer false suggestions, i.e. merges that would result in an erroneous model. Because of this, the legality of each merge must be manually checked to ensure the validity of the model. The merging results in a *control model* which contains the functionality of the original test cases, but without the information encoded into variables. The merged model, variable models and initialization models are then passed through parallel composition, which creates an executable test model.

While the test model obtained this way is usable, it may pose difficulties for test generation. That is because the model is likely to contain many paths leading to deadlocks, i.e. states with no outgoing transitions, resulting either from a denied precondition or the end of a test case that could not be merged anywhere. The model may be cleaned by removing all the dead paths, but this is not always a good idea. If the test cases could be looped back into themselves and deadlocks occur only or mostly in places where a precondition fails, the clean-up procedure should be safe to perform. Conversely, if many test cases ended in unique states and caused deadlocks at the end, the clean-up could remove relevant functionality. In this case the model may be better left as-is, and the test generation algorithm must take care not to guide the execution toward a deadlock prematurely.

Figure 5 shows the model obtained from our example test cases. Merges have been performed at matching actions and state labels. Adding the initialization models and the variable model in parallel composition results in the usable model depicted in Figure 6. It would seem that in this case the cleaned model (Figure 7) would be more useful, since the dead end on the right likely serves no practical purpose; it depicts a situation where a new message is created with one already on its way, causing the preconditions to block its sending.

4 Case Studies

We have performed two small scale case studies to test our synthesizing methodology. The original sequences were linear keyword level test cases picked from a much larger

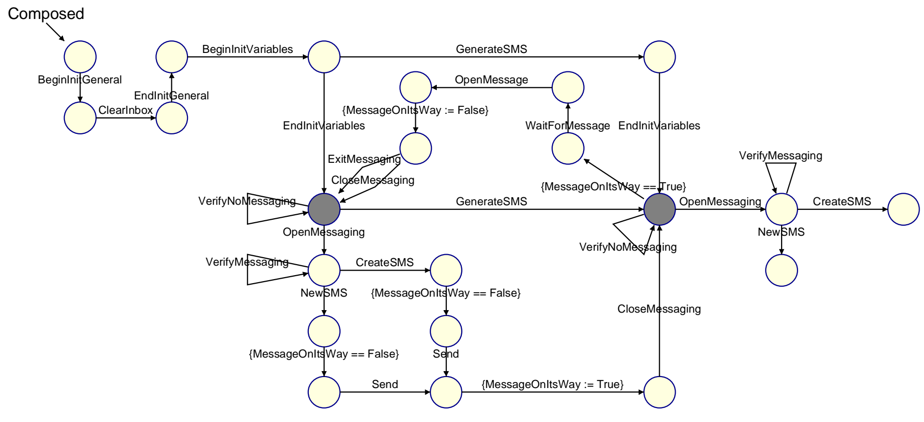

Fig. 6. The example model after parallel composition.

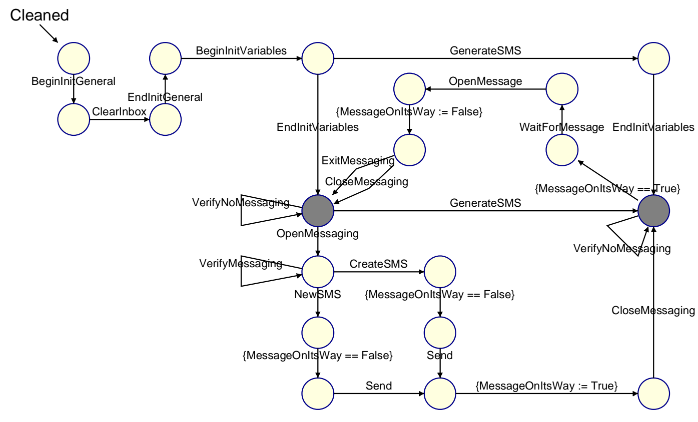

Fig. 7. The final, cleaned model.

set of test cases for S60 applications developed by one of our industrial partners. The first case study used seven test cases for the Phonebook application. The second one had nine for the Messaging application, concentrating on short and multimedia messages (SMS and MMS). Both case studies used the same set of 30 keywords. The Phonebook test cases had 193 actions altogether, the Messaging test cases 363. Three of the test cases for the Messaging case study can be seen in Figure 8, with some changes made for readability and to adapt them for a single phone.

Both case studies used the same set of keywords, which we were already familiar with from our earlier work. Giving keywords their weights was therefore easily done, though the values were somewhat arbitrary; we had yet to perform enough experiments to find the best values. The Phonebook case proved to require seven variables, six to hold information about existing contacts and groups and one for incoming messages. The Messaging case required six variables, two for the existence of messages and reports and the rest for various settings. The first case labeled the idle state and the contacts and groups screens, the latter labeled the idle state and the screens for SMS and MMS writing.

After the merge, the Phonebook model had 126 and the Messaging model 192 states. Parallel composition and cleanup brought state counts to 12523 and 2327, respectively. The Phonebook case shows the potentially exponential growth caused by variables. This happened because the variables controlled relatively small portions of the model and had little to do with each other. Conversely, the variables in the Messaging case were interconnected to some degree, and affected control to a much greater extent; for example, many individual test cases specified certain settings before sending a message. As a result, large portions of the control model were reachable only with certain variable values. Figure 9 shows an overview of the final Messaging model, illustrating its scope and complexity. Although the models are too large for human understanding, their size is not a problem for our automated test generation tools.

The quality of the final models appeared to be comparable to the test models in our test model library [9] created by hand from scratch, although not quite equal to them. The synthesized models contained less functionality, but this was a result of the original choice of test cases, not a failing of the method itself. A notable difference was the higher granularity of the synthesized models: often actions which could be performed separately in hand-made models were forcibly chained together in synthesized ones. However, this tendency did not seem to reach truly detrimental levels, and the number of possible action sequences was still magnitudes higher than in the original linear test cases. The final difference between the synthesized models and our old models was that keyword level test cases naturally became a single keyword level model, not a combination of keyword and action word level models as in our model library. The action word level might be added using the bottom-up modeling technique presented in [6]. Presumably action word level test cases could be combined into an action word level model and the action words then refined as in original test cases, though we have yet to attempt that.

In both case studies, most of the effort during the synthesizing process went into variable definition and integration. In the Phonebook case, this was mostly manual work: the variables were simple, but referenced often. With Messaging the situation

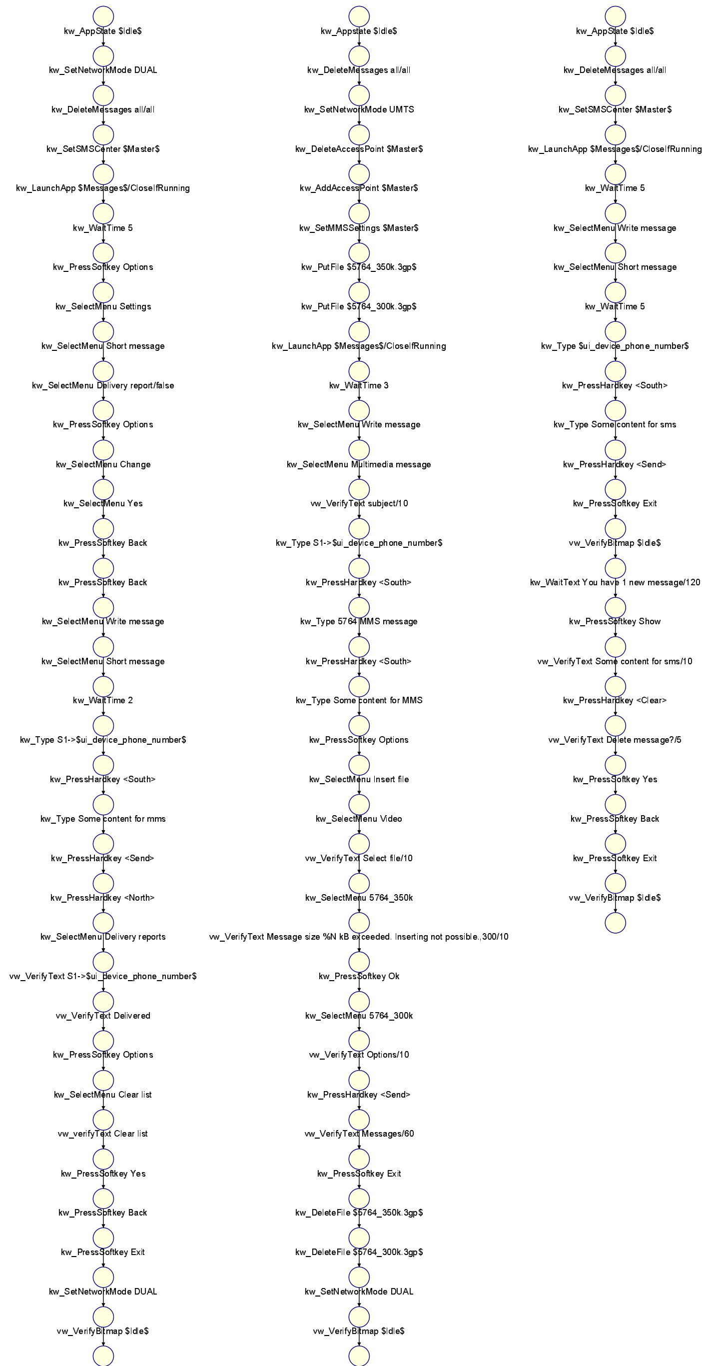

Fig. 8. Three of the nine Messaging test cases.

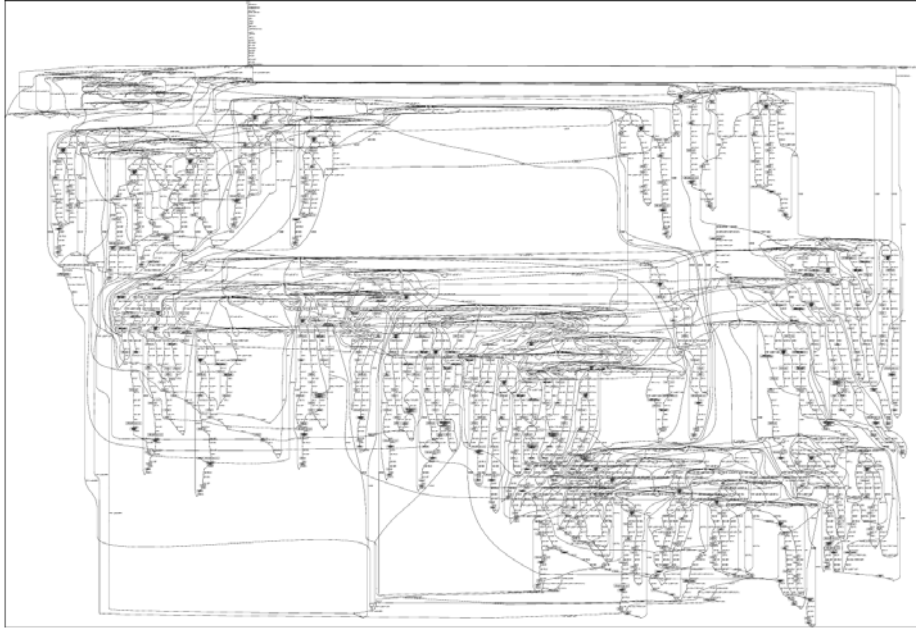

Fig. 9. The final Messaging model.

was different. There much time was spent in deciding what exactly should be modeled into variables, and how exactly would they be integrated into the test cases. Placing the pre- and postconditions also took considerable time, mostly because the complexity of the variables demanded great care in integrating them into control. We found merging to be relatively easy, but it might pose more difficulties to someone not used to test modeling. It definitely requires some understanding of the implemented variables, which implies that the whole process might be best performed by a single person.

Both of the case studies were performed by a single person and each required less than a day to complete. It seems quite reasonable to us that with good tools a person familiar with the process could synthesize a model of considerably greater size within a single day. That would be notably faster than creating a comparable test model from scratch, and would not require a similar expertise in modeling. Fortunately the most time-consuming phase, variable definition and integration, seems likely to scale reasonably well with the number of test cases (probably linear effort or less). The least scalable phase by far is merging of the component models (potentially quadratic or even exponential effort), which at least might be fully automatable.

5 Discussion

In this paper we have described an approach for synthesizing test models from test cases. In addition, we presented the results of two small case studies where the approach

was applied for creating test models from existing test cases in the domain of S60 GUI testing. The synthesis is semi-automatic and thus requires user interaction to achieve useful results. A tool supporting this interaction was also sketched.

Our approach is domain-specific in the sense that the set of keywords and the corresponding weight values must be decided based on the domain knowledge. In our case studies this was easy because the same person who had built our model library conducted the experiments. However, the other phases of the synthesis process should be applicable also in other contexts.

There exists a large body of knowledge about the synthesis process. While most, if not all, of the existing approaches have been originally developed for design, analysis and code generation purposes, they may be useful for test model synthesis also. Amyot and Eberlein have compared twenty-six solutions for constructing design models from scenarios [14]. Moreover, Liang, Dingel, and Diskin have developed comparison criteria for comparing different algorithms and applied the criteria to compare twenty-one different approaches [15]. However, it seems that domain knowledge can improve the synthesis; we first experimented with a more generic approach [16], but decided to develop our own to better fit the needs of our context. An extensive study would be needed to analyze the other existing approaches for their applicability to test model creation, but this lies outside the scope of this paper.

Some of the currently manual phases in our synthesizing process might be automated, most notably initialization and parts of modeling of variables. The action weights and state labels must be set manually. The defining and integration of variables also requires user input, but actual variable models can be created automatically. It might also be possible to automate merging totally, not just finding the potential merge points. In the two case studies, potential merge points occurring at state labels were always mergeable; this seems likely to be a general rule, as long as the labels have been placed well. The merge points based on action sequences varied, some being mergeable and others not. However, in these cases the sequence merges did nothing that could not have been replicated with well-placed state labels. Based on these observations, it might be possible to automate the merging to always merge at labels and disregard sequences altogether, but more testing is required before implementing such changes.

Although the most work-intensive part of the process, the creation and integration of variables, cannot be truly automated, it could be substantially eased by proper tools. These should offer both an easy way to define variables, preferably hiding the models altogether, and a simple method for setting pre- and postconditions. Some algorithm for suggesting potential variables would be a highly useful feature, but difficult to design.

In the future, in addition to developing tool support, there is also the need to conduct wider case studies and to compare the test coverage that can be achieved with handcrafted versus synthesized test models in actual on-line test generation.

Acknowledgements

This paper reports results of research funded by the Finnish Funding Agency for Technology and Innovation (TEKES), Nokia, Conformiq Software, F-Secure, and Plen-

ware, as well as the Academy of Finland (grant number 121012). For details, see http://practise.cs.tut.fi/project.php?project=tema.

References

1. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann (2007)
2. Kaner, C., Bach, J., Pettichord, B.: Lessons Learned in Software Testing: A Context-Driven Approach. Wiley (2001)
3. Robinson, H.: Obstacles and opportunities for model-based testing in an industrial software environment. In: Proceedings of the 1st European Conference on Model-Driven Software Engineering, Nuremberg, Germany (2003) 118–127
4. Hartman, A.: AGEDIS project final report. Available at http://www.agedis.de/documents/FinalPublicReport%28D1.6%29.PDF (2004) Cited June 2008.
5. S60. (http://www.s60.com. Cited June 2008)
6. 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
7. Jääskeläinen, A., Katara, M., Kervinen, A., Heiskanen, H., Maunumaa, M., Pääkkönen, T.: Model-based testing service on the web. In: Proceedings of the the 20th IFIP Int. Conference on Testing of Communicating Systems and the 8th Int. Workshop on Formal Approaches to Testing of Software (TESTCOM/FATES 2008). Number 5047 in Lecture Notes in Computer Science. Springer (2008) 38–53
8. Nachmanson, L., Veanes, M., Schulte, W., Tillmann, N., Grieskamp, W.: Optimal strategies for testing nondeterministic systems. In: ISSTA '04: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis, Boston, MA, USA, ACM (2004) 55–64
9. Jääskeläinen, A., Kervinen, A., Katara, M.: Creating a test model library for GUI testing of smartphone applications. In: Proceedings of the 8th International Conference on Quality Software (QSIQ 2008), IEEE Computer Society (2008) 276–282
10. Fewster, M., Graham, D.: Software Test Automation: Effective use of test execution tools. Addison–Wesley (1999)
11. Buwalda, H.: Action figures. STQE Magazine, March/April 2003 (2003) 42–47
12. Hansen, H., Virtanen, H., Valmari, A.: Merging state-based and action-based verification. In: Proceedings of ACSD 2003, the Third International Conference on Application of Concurrency to System Design, IEEE (2003) 150–156
13. Virtanen, H., Hansen, H., Valmari, A., Nieminen, J., Erkkilä, T.: Tampere verification tool. In: Proceedings of TACAS 2004, Tools and Algorithms for the Construction and Analysis of Systems, the 10th International Conference. Volume 2988 of LNCS. Springer-Verlag (2004) 153–157
14. Amyot, D., Eberlein, A.: An evaluation of scenario notations and construction approaches for telecommunication systems development. Telecommunication Systems **24** (2003) 61–94
15. Liang, H., Dingel, J., Diskin, Z.: A comparative survey of scenario-based to state-based model synthesis approaches. In: Proceedings of the 2006 International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools (SCESM'06). (2006) 5–12
16. Mäkinen, E., Systä, T.: MAS - an interactive synthesizer to support behavioral modeling in UML. In: Proceedings of the 23rd International Conference on Software Engineering, IEEE Computer Society Press (2001) 15–24