Test-based model construction

Jon Gjengset
for Oracle Labs, Brisbane
and Bond University

April 1st, 2011

Abstract

Model checking and model-based test generation are both well proven techniques for improving the quality assurance process for software. Unfortunately, constructing the necessary models requires expertise beyond that held by most QA teams. Training these teams in abstract model construction would require much in terms of time, cost and effort, and might not be a viable option for many businesses when held up against the potential gains.

This paper explores the process of constructing a model solely from a pre-existing test suite of an application. More specifically, it examines the viability of automating this process to give QA teams a tool for automatically enhancing their test suites.

Due to the time constraints of this project, the goal of this paper is only to examine the requirements for such automatic generation, and not to actually implement any tools to achieve this.

Contents

1 Executive summary **2**

2 Project Background **3**
2.1 Modelling . . . 3
2.2 Model-checking . . . 4
2.3 Test generation . . . 4

3 Introduction **5**
3.1 Project execution . . . 5

4 Related literature **5**

5 Metrics and observations **5**
5.1 Modelled applications . . . 6
5.1.1 touch . . . 6
5.1.2 WiFi driver . . . 6
5.1.3 NWAM daemon . . . 7
5.1.4 tbase.c . . . 8
5.1.5 svc.startd . . . 9
5.2 General notes . . . 10
5.2.1 Stages . . . 10
5.2.2 Enumeration . . . 12
5.2.3 Abstraction . . . 12

6 Model usefulness **12**
6.1 Generated tests . . . 13
6.2 SAL . . . 13

7 Comparison of model developing bases **13**

8 Conclusion **14**

Appendices **16**

A Model code **16**
A.1 Touch . . . 16
A.2 WiFi driver . . . 17
A.3 tbase.c . . . 19
A.4 svc.startd . . . 20

B Notes **25**
B.1 touch . . . 25
B.2 WiFi driver . . . 26
B.2.1 Model description . . . 27
B.3 NWAM . . . 29
B.4 tbase.c . . . 31
B.5 svc.startd . . . 32

1 Executive summary

This paper explores a method for automatically constructing software models from an applications test suite. This can be used to improve the quality of application test suites by using model-based test generation to generate additional tests. This is attractive since better test suites generally guarantee applications with fewer faults. Writing extensive test suites is a time-consuming task, and having quality assurance teams writing them manually often lead to incomplete or erroneous tests. The approach outlined in this report would allow a quality assurance team to automatically augment their existing test suites, thereby covering more of the application's behaviour. It works by first parsing the test cases into an abstract model, and then using model-based test generation to automatically construct new test cases.

In broad strokes, the algorithm is as follows:

1. Run the existing test suite through a translator to get rid of language- and framework specific code.
2. Take the stripped code and run it through an interpreter which replaces every command with how it changes the state of the program.
3. Use the model generator to convert the set of state changes from (2) into a model.
4. Use a test generator to construct new test runs from the model.
5. Feed the generated test runs into a test adapter which translates the runs into executable code (the reverse of (1)).

This report includes observations gathered from the development of models for five applications. The development of these models was done manually to assess the tools necessary in order to automate this process.

After an initial survey, this paper concludes that although this may prove a viable technique to supplement writing traditional tests, it does require a large initial development effort in order to work with any application. This is due to the fact that for any given application, two quite complex, application-specific components would have to be written: the translator and the interpreter. These are needed to allow the model generator to understand the semantic meaning of each test case, and have to be written specifically for the test suite in question. Thus, the method as outlined in this document is only recommended for applications with a very large test suite where it is infeasible to write software models from scratch.

Furthermore, it has proven difficult to model very configuration-dependent applications. The reason for this is that models cannot easily represent programs that exhibit different, and in some cases directly opposite, behaviour given the exact same input. Configuration-heavy applications inherently behave differently depending on their current configuration, and as such, their behaviour cannot be accurately expressed using a single model. One could solve this by constructing multiple models; one per configuration, but for most applications this is impossible since they often have an infinite number of possible configurations. Therefore, the method outlined above is not applicable to many such programs.

It is recommended that readers not familiar with the underlying concepts of modelling and model-based testing read section 2 for an introduction to the field.

2 Project Background

This first section will give an introduction to the underlying concepts of model based testing, as well as give references to other works in the field that can be useful to newcomers to the field.

Today, the main way of testing the correctness of computer programs is to write test suites consisting of multiple test cases, each exercising one behaviour of the system being tested and checking that the resulting output or system state is "correct". This approach has several weaknesses; mainly the fact that a single test can only test one behaviour. Therefore, a great number of tests are required to exhaustively test any system. In fact, since many systems have a near infinite amount of possible unique executions, it is in many cases impossible to verify the correctness of a program using only test cases.

Software modelling is an emerging field in the area of software testing and verification, in which a system is represented as a set of states and guarded transitions between these. This state graph can then be used together with a symbolic model checker to either verify that certain properties hold for the modelled system (for instance that no action in a banking system will allow the balance to drop below zero), or to generate execution traces through that graph. An execution trace contains a series of inputs and expected outputs which represent one run of the program (i.e. a test case).

It is not within the scope of this report to go into the details of how model-based test generation works, but a brief overview is given below of what models are, and how they can be used for software verification and test generation.

2.1 Modelling

Formally, a model consists of three sets: the variables that define the state of the program, the possible initial values for those variables, and finally, the conditions that may cause any of the state variables to change its value. Less formally, a model can be envisioned as a graph where each node represents an assignment of values to all of the values, and where each edge from a node represents actions that change the values of those variables. Furthermore, each such action can be guarded so that an edge can only be traversed if certain conditions are satisfied.

This might be easier to explain with an example. Let us model a simple light switch using the modelling language SAL (outlined in [de Moura et al., 2003]):

```
1  switch: CONTEXT =
2  BEGIN
3    Operation : TYPE = {turnOn, turnOff}
4
5    switch: MODULE =
6    BEGIN
7      INPUT op : Operation
8      LOCAL on : boolean
9
10     INITIALIZATION
11       on = false;
12
13     TRANSITION [
14         (on = false AND op = turnOn) --> on' = true;
15       []
16         (on = true AND op = turnOff) --> on' = false;
17       ]
```

```
18    END;
19  END
```

In the code above, there is a single state variable, the boolean *on*, and a single input *op* which is either "turnOn" or "turnOff". This translates roughly to the following simple graph with off as the initial state:

```
(off) --[turnOn]--> (on)
  ^----[turnOff]-----+
```

This is the model.

From here, two things can be done: model-checking and test generation.

2.2 Model-checking

Although model checking is not used in this paper, it can be useful to get a better understanding of the merits of software modelling. In model-checking, one defines a set of properties that should always hold regardless of the current state of the system and the sequence of inputs. The model checker then, in theory, visits every reachable state, and verifies that the property holds on that state. Practically though, many optimisations are made, but this is outside the scope of this introduction. In SAL, the properties are expressed using Linear Temporal Logic, which is explained thoroughly in [Voronkov, 2010].

If the model accurately represents the system under test (SUT) and the properties do indeed fully describe the behaviour that should be verified, then a model checker can prove the correctness of the SUT. Unfortunately, the models are often either over-simplified or the properties too vague, causing the model checker to report success even though some behaviours were in fact erroneous.

For a more thorough explanation of model-checking, see [Ammann et al., 2002].

2.3 Test generation

Model-based test generation is quite different from model-checking in that it does not verify the correctness of a program, but rather generates tests that exercise particular interesting behaviours. First, a set of reachability goals are declared. A reachability goal is a boolean value which the test generator will attempt to make true. The generator will explore the graph to find sequences of inputs that cause every reachability goal to be satisfied at least once. The resulting input sequences represent executions of the program that each test one or more aspects of the system, and can be used as test cases for the system using a test adapter. These test cases will fully test all the properties expressed by the reachability goals.

In the above light switch example, one could add a reachability goal *wasTurnedOn* that is set to true upon reaching the *turnOn* state. Running the test generator on the model will then generate a test case that consists at least of the input *turnOn*, as this is required to satisfy the goal.

For a more thorough explanation of model-based testing, see [Utting et al., 2006].

3 Introduction

Unfortunately, the development of software models requires a set of skills that few quality assurance teams have today; especially if they are to be more useful than the test cases they are to replace or supplement. The purpose of this paper is to explore the possibility of creating useful abstract models solely from existing test cases, and more importantly, the feasibility of doing so automatically. Automating this process would remove the need to educate QA teams in the development of models, and could allow for automatic augmentation of a test suite with little or no involvement from the QA teams through model-based test generation.

3.1 Project execution

As the objective of the project is to determine the feasibility of developing models based on test suites, it makes sense to examine if the approach will work on different types of applications. Therefore, the test suites of three types of applications have been examined, and manually designed models for: command line utilities, drivers and servers/daemons. During the development of these models, the techniques required to construct these models have been noted down, and a log has been kept of challenges encountered in the process. A record of metrics such as man-hours, the complexity of the models and lines of code has also been kept. Throughout the process, the modelling has been done algorithmically (i.e. using as little reasoning as possible) to identify problems specifically related to making the construction automated.

All the models have been written using the modelling language SAL from SRI. [de Moura et al., 2003] gives an in-depth manual for the SAL language. SAL was chosen since it has a very straightforward syntax and is readily available. It is also fairly mature, and comes with both a symbolic model checker and a test generator. SAL might not be suitable for this kind of application however, as it is not trivial to interface with the SAL tools. Any would-be test adapter would have to parse the textual output from the SAL test generator, rather than use native API calls. NModel or PyModel might therefore be more suited for larger-scale applications of model-based test generation.

The project was carried out over 12 weeks in the first quarter of 2011 with the help of Lian Li at Oracle Labs, Brisbane.

4 Related literature

The only paper directly discussing constructing software models from test cases is [Jääskeläinen et al., 2009]. However, that paper only explores the process of taking a test suite where the exact state-changes incurred by each command in each test case are known. In reality, most of the problems related to generating a model are not related to the generation itself, but rather to understanding the "meaning" of the commands in each test case.

5 Metrics and observations

Over the course of this project, five applications were modelled, one of which was abandoned (the reasons why are given below). The five applications were chosen because they represent quite different application types: command line utilities, drivers and daemons.

The metrics on the work effort required, as well as the final size and complexity of the models are given in table 1 below.

During the writing of this paper, an informal work log was also kept while developing each model. These should be read to get a full understanding of the developed models, and can be viewed in full in Appendix B. The final SAL code for each model is listed in Appendix A.

NOTE: Many of the test suites below are taken from the Solaris Test Framework, and are available for download at `http://dlc.sun.com/osol/test/downloads/current/`.

Model	Size	Hours	Transitions	Variables	Test suite size
touch	80 LOC	5	6	0	663 LOC
wifi-driver	103 LOC	3	14	4	716 LOC
NWAM daemon	–	–	–	–	1990 LOC
tbase driver	56 LOC	40 mins	8	5	886 LOC
svc.startd	175 LOC	5	18	5	6629 LOC

Table 1: Model metrics

5.1 Modelled applications

5.1.1 touch

Description The first application to be modelled was the UNIX "touch" command. It is a utility that lets a user modify the time-stamps of a file conveniently from the command line.

Tests The test code for touch can be found in the "coreutils" package available here under tests/touch/. It deals mainly with tests for corner-case behaviour, such as what happens if an empty file or a dangling symbolic link is touched, and not with "normal" behaviour.

Model The model for touch becomes slightly awkward since touch keeps no state between invocations. It only relates a set of input values to certain outputs (e.g. if the file is no writeable, and not owned by the user, an appropriate error should be returned). Furthermore, it is worth noting that the execution environment needs to be made explicit as inputs. For example, whether the file is empty, is a directory or a FIFO file are not inputs in the classic sense, but they do affect the behaviour of the application, and so must be considered input values.

Notes The touch model uses no state variables, as evident in the model code (Appendix A.1). The goals variables are only there for the test generator, but do not actually create any state.

5.1.2 WiFi driver

Description This test suite is not for one particular driver, but rather a generic test suite for the Ad-Hoc mode of wireless network interface drivers.

Tests The tests can be found in usr/src/suites/net/driver-wifi/tests/ibss of the Solaris Test Framework WiFi Test Suite (stcnv-wifi). As opposed to the touch tests, these test cases do actually test normal behaviour such as that a normal connect operation will succeed, and not just the odd cases.

The WiFi tests often test the same behaviour twice, once with the local host as the creator of the network, and one with the remote party as the creator. This is something that should be enumerated (See 5.2.2), and is thus not represented by goals in the model.

Model This is the first model where state variables are being used. Since the tests often try multiple operations in succession (connect, disconnect, reconnect), whether the interface is currently connected or not, whether another party is connected, etc., needs to be kept track of.

The transitions are easily extracted from the test cases as they are written in a very readable and clear style. For instance, in tp_ibss_003.ksh an Ad-Hoc network is created, a remote client is connected, the local client is disconnected, and then attempts to reconnect to see that the network does not go down even though one party disconnected. This maps quite nicely in the model to the operations *connect*, *remoteConnect*, *disconnect*, *connect*, and is in the model represented as the goal *goalReconnect*.

Notes Although this application is more complex than the touch program, its stateful nature makes it easier to model. The WiFi driver also exposes the first major hurdles towards automatically generating a model. The following is an excerpt from the work log for this model (Appendix B.2).

> [...] the two test cases tp_..._004.ksh and tp_..._005.ksh test essentially the same thing - does the device driver successfully send and receive packets, but does so using two different tools (ping and runuperf). A model synthesiser would need the ability to "understand" this, and merge them into a single aspect of the model (i.e. a traffic goal). Trying out both these types of traffic should be left up to the adapter, as the differentiation does not exist in the model.

Some configuration options do not result in different behaviour from the models perspective, but they do affect the internal workings of the application. For instance, one can use many different security settings when connecting to a network, but no matter what the settings are, the state only changes from *not connected* to *connected* if the operation is successful. This problem is further discussed in section 5.2.2. The test adapter (i.e. the code that translates between the generated test runs and executable code; see 5.2.1 under Execution) should be able to enumerate the possible settings, and try each one when the model says to try one.

5.1.3 NWAM daemon

Description The Network Auto-Magic (NWAM) daemon is a Solaris tool for managing network interfaces automatically. It allows a system administrator to set up rules as to what interfaces should be brought up in response to outside events, and enable or disable location profiles (proxy settings, default gateway, etc) based on the network state. For instance, a system administration could make the computer use a particular HTTP Proxy when it is given the IP 192.168.30.62. A more thorough description can be found on the Oracle NWAM support website.

Tests The test suite is located in usr/src/suites/net/nwam/tests of the Solaris Test Framework. Unfortunately, at this moment, the NWAM tests are not publicly available, but I have been informed that they are available from Oracle. Since the test suite is quite substantial, only on the event-handling behaviour of the program was modelled. The event specific test cases are located in the sub-directory event/ and was chosen since event-driven programs are usually very suited for modelling as the events map quite nicely to inputs that trigger state changes.

The tests are mainly concerned with bringing up network interfaces with certain rules attached to them, and then testing that the corresponding rule was applied. For instance, one test case could contain a rule stating that an interface should be not be started by default. The test case will then run the svc.startd daemon and verify that the server is not running after svc.startd is started.

Model No model was written for NWAM. See the notes below for an explanation.

Notes This application proved very hard to model efficiently. The reason for this is that its behaviour is inherently very configuration-dependent; two different configurations could make the application behave differently when given the same input. Since a model describes the behaviour of an application, or more specifically, how each input affects the state of the application, one would effectively need one model for every single configuration, but this is not viable when the number of possible configurations are infinite. Using test cases, this problem is non-existent since the tests set up one specific configuration environment, and then check that it exhibits the wanted behaviour. Eventually, it was decided that since a good way of modelling this with a limited number of models could not be found, this application would be abandoned.

For a more detailed look at what caused this to fail, refer to the notes taken during the modelling process in Appendix B.3.

5.1.4 tbase.c

Description Like the WiFi driver test suite, this is also a generic test suite, and does not apply to any particular application. tbase.c is a base test suite for kernel device drivers.

Tests The test suite code can be found in the Linux Test Project under testcases/-kernel/device-drivers/base/tbase/tbase.c.

The tests contained in tbase.c mainly test the registering and removal of device major and minor numbers in the UNIX device table. The tests are all fairly small, and test for instance that getting a reference to a device increments the reference count for that device node or that creating a new device does indeed return a new device number.

Model Since this model turned out to be fairly small, no goals were added to allow more time for modelling another large application. Nevertheless, the model should incorporate all the behaviour expressed through the test cases in tbase.c. Essentially, all that was required for this model was to look through the test cases, find all the method calls and turn them into INPUT operations, and then figure out how they changed the state of the program. With tbase, the state space was fairly small, and as such, each transition was also quite small.

Notes In the tbase.c test cases, another important aspect of test cases that are tested well in the model was found. The following is an excerpt from the work log (Appendix B.4):

> I have also noticed that some operations have a no-op [An operation that does nothing] end result because they execute corresponding create and close operations (e.g. test_create_file runs driver_create_file, and then driver_remove_file just to test that running driver_create_file will actually work). The problem with this is that no program state has changed as a result of the operation, and the running of driver_create_file does not seem to change the state of the program, meaning it does not make sense to put this in the model. However, these operations should still be tested.

In this case, the test case could be split into two transitions, one which creates the file, and another which closes the application, but the question becomes: How can this be detected automatically?

5.1.5 svc.startd

Description svc.startd is the master restart daemon for the Service Management Facility, the Solaris module that handles services and daemons. It ensures that services are brought up when they should, restarted if they crash, brought down if they fail, and so on. Further information can be found in the MAN pages.

Tests The tests are located in the Solaris Test Framework SMF Test Suite (stcnv-smf) under usr/src/suites/os/smf/tests/svc.startd/tests/. Again, due to the massive size of this test suite, only the tests in "methods/" where modelled. The "methods" subset was chosen since each svc.startd method corresponds nicely to different "operation" inputs to the model.

This test suite was the most extensive one used in this project, but turned out not to be too hard to model. The biggest obstacle was understanding the meaning of all the function calls in the test suite since these were not well documented. Once understood however, the modelling process was quite straight-forward. The test suite contained tests both for normal behaviour and corner cases, and thus provided the basis for a quite feature-complete model.

Model In the case of svc.startd, the model represents only the state of a single service that svc.startd is managing. This abstraction is useful since the behaviour of startd is independent of the number of applications it manages. The only thing that matters to startd is what state the application is in, and what command is given.

It is worth noting here that only a single state variable is really needed here; one which represents what state the service is in (online, disabled, maintenance...). The other state variables (starting with "onlyRetry") are only there to handle the case where svc.startd attempts to run a command multiple times before giving up (discussed further below).

The output of the model is used to tell the test adapter what command it should verify is run, similar to what is done in several of the test cases.

Notes In the case of svc.startd, the only state that really had to be kept was whether the current service was running or not. The added complexity came from the fact that the test cases also captured behaviour where, if an action failed, it was retried once before being sent to maintenance mode. This retry operation required there to be a guard on every other transition to prevent SAL from choosing another transition than the retry operation of the previous command failed. This is yet another use case that an automatic model generator will have to detect and respond to.

Also in this application, the configuration "problem" is encountered, although this time in a slightly different form. For instance, svc.startd responds differently to a *stop* command depending on whether a stop behaviour is defined in the service's initialisation file. This is environment configuration rather than application configuration, but it still alters the behaviour of the program for certain inputs. Therefore, a way of representing it in the model will have to be developed, or a single model for every single configuration option would be needed as with NWAM. In the case of svc.startd, this was solved by adding another INPUT variable that states whether a stop directive exists. This way, the test generator can easily differentiate between the cases and modify the environment accordingly. This works in this case, but will probably not scale well if there are lots of external factors affecting the application's behaviour as was the case with NWAM.

Another downside to the way svc.startd is modelled is that *initialState*, *stopExists*, *commandTimeout* and *commandSucceeds* had to be modelled as inputs. This causes problems when doing the test generation since they now contain sequences of repetitive calls to *enable* where the only difference is different values for the variables mentioned above, even though some of them are irrelevant to the behaviour of *enable*. This is not a blocking problem, but a nuisance since it unnecessarily bloats the generated test runs.

5.2 General notes

After modelling all five applications, some commonality in how the models are developed becomes apparent. The process seems to be somewhat easily divided into five distinct stages:

5.2.1 Stages

Translation

This step does not apply as much to manual modelling, but will certainly be vital in doing any kind of automatic model generation. Test suites are written in a plethora of languages and using many very different testing frameworks. To avoid having to write a model generator for every combination of these, translators will have to be written to convert all tests into a common language and style that the generator operates from. This does not have to be an abstract, special-purpose language; the important thing is that all the test cases have the same syntax.

As an example, consider the following bash test code:

```
1  set_up eth0;
2  connect eth0 ``MyNetwork'';
3  verify_connected eth0;
```

and this Java snippet

```
1  public boolean testConnect() {
2    WifiInterface eth0 = new WifiInterface(``eth0'');
```

```
3       eth0.connect(``MyNetwork'');
4       return !eth0.isConnected();
5     }
```

They both test the same thing, and affect the state the same way, but are very different syntactically. The translation stage would allow them both to be translated into this more abstract form:

```
1   eth0 # There is something called eth0
2   connect eth0 # There is a command connect that changes eth0
3   eth0.connected == connected # There is a check that eth0 is
        connected
```

Interpretation

Next, the test cases need to somehow be understood by the model generator. There has to be established some connection between a command and its effect on the overall state of the program. For instance, what state variables change as a result of a *connect* command in any given application? A grammar has to be created that enables the interpreter to convert test case commands into state-transitions.

To illustrate what such a grammar will achieve, consider a command called *connect*. To an interpreter, *connect* is just a series of letters with no meaning. The grammar will allow it to replace connect with something like this: $connected = false; connect; connected = true$; revealing the exact state change associated with that command. After this stage has been completed, sufficient information has been extracted to utilise the technique outlined in [Jääskeläinen et al., 2009].

Modelling

Once the meaning of each test case has been established by the interpretation stage, the model generator will have to run through all the commands and combine them into one grand model that encompasses all the state transitions expressed in the tests. It also has to insert appropriate reachability goals for every currently existing test case. Furthermore, the model will have to do enumeration (see 5.2.2) and abstraction inference (see 5.2.3)

Generation

After the model has been generated, and the reachability goals set, the model has to be run through a test generator. These are available for most modelling languages already, and simply generate test runs that hit every defined goal at least once as explained in section 2.3.

Executing

Finally, the generated test cases need to somehow be executed against the application, or converted into executable test cases. For this, a test harness or test adapter will be needed that can understand both the abstract test run syntax, as well as an executable programming language. The adapter will also have to apply all the defined enumerations (see 5.2.2) and pick values for any abstracted properties.

For instance, the test generator may output something like this:

```
1   Step 0:
2   --- Input Variables (assignments) ---
3   op = connect
4   --- System Variables (assignments) ---
```

```
5    up = none
6    connected = false
7    ------------------------------
8    Step 1:
9    --- Input Variables (assignments) ---
10   op = show
11   --- System Variables (assignments) ---
12   up = this
13   connected = true
```

At this point, the test generator should run the *connect* command, and check that the interface is in fact connected afterwards. More importantly, it should run connect (and any subsequent command) once for every possible security setting, since this was one of the configuration options that should be enumerated.

5.2.2 Enumeration

Some configuration options do not affect the behaviour of the application, but do still affect the workings of the application on the implementation level. An example of this could be the *connect* command for the WiFi driver. A network may have many different security settings (e.g. none, WEP, WPA-TKIP, WPA2-AES...), but these do not change the behaviour of *connect*. After *connect*, the interface is either connected or it is not. Thus, this is not something that should go in the model considering the behaviour is the same. Instead, this can be left up to the test adapter. It should "understand" that whenever it sees *connect*, it should rerun the entire test run using all the different possible values for the security settings. The downside of this is that the interpreter that parses the test suite initially will have to detect these enumerated options, and somehow store the knowledge that certain options should be enumerated when it comes to generating executable test cases.

5.2.3 Abstraction

Another important task of the interpreter will be to partition any boundless options or arguments into a finite number of equivalent parts. For instance, touch allows the user to specify the time it should set the modified time of the file it operates on to. This option is pretty free-form, and can be anything from "-5 years" to "now + 1 week 2 seconds". Naturally, the model cannot be made to try every possible value, as there are an infinite number of them. What should be done instead is to partition these into "past", "now" and "future", or even further into "past-year", "past-week", etc., to make the model test more of the behaviour.

6 Model usefulness

Given a translator, an interpreter and a generator, are the generated test cases useful? It depends on what tests the test suite contains from before. If the test suite has a very high degree of coverage, the amount of additional test cases the model can provide is limited. Also, the model can only test the features that are expressed through the test suite. If a feature is not in the test suite, then the model, being constructed only from the tests, will naturally not be able to test this feature either.

Furthermore, the number of applications this model generation will work for is severely limited by the configuration problem that was encountered when modelling the NWAM application. If configuration-heavy applications are close to impossible to model well, then the test suite cannot be expanded - neither manually nor automatically.

6.1 Generated tests

Due to time constraints, test generation was not part of the scope for this report. Nonetheless, tests were generated for some of the models, and the following observations were made:

- In the "touch" test suite, few new tests were generated since no state is kept, and thus the order of commands does not matter.
- For the WiFi driver, the generated test run includes running a scan after a suspend/resume cycle, which the original test suite did not. This is a very sensible test to include, yet it was left out by the original test authors.
- Another test generated for the WiFi driver that was not in the original test suite is connect-suspend-resume-disconnect. Again, this is something that should be in the test suite (i.e. does a connection survive a suspend/resume), but which was missing.

A note on svc.startd The generated tests for svc.startd are quite hard to analyse since they contain a lot of duplicated calls (multiple *enable* calls in sequence for instance), making them very large. Furthermore, since the model ended up being quite a lot more complex than any of the others, there are some shortcomings that also become visible in the test runs. For instance, in one part of a test, *disable* is called after a core dump, which does not make sense in a real-world scenario. Due to the limited time available, and since it is outside of the scope of this report, no further detailed analysis was done on the svc.startd tests.

6.2 SAL

The model test generator provided with SAL by default only tries to generate test runs so that every defined goal is hit at least once. This does mean that in some cases, it might only generate the exact tests that are already present, since the goals were derived from those. More often than not however, the test generator also ends up constructing more complex test runs that involve many transitions, or more importantly, orders of transitions, that were never tested in the original suite. In order to force this behaviour for any test generator, a reachability goal could be added in every transition, causing the generator to at least hit every transition once. This can be further extended to force it to check different possible orderings of commands, but this has not been examined during this project.

7 Comparison of model developing bases

Due to the limited time available, no models have been created from scratch (i.e. from the specifications rather than from the test suites) to do any direct comparison. However, it is still possible to do some reasoning on the merits of each approach:

From test cases

- Limited to modelling the behaviour of the features present in the test cases
- Can potentially be constructed automatically
- Should be regenerated every time tests change

From specification

- If the application is successfully tested by the model, the application is correct according to the specification
- Must be constructed by hand
- Implementation-specific bugs may not be caught
- Will be fairly static, as specification changes less frequently than tests and code
- Should be written by QA teams, or someone else with much knowledge of specification, but little of code to avoid bias

From code

- Has a higher probability of catching implementation-specific bugs
- If the application is successfully tested by the model, it is not guaranteed to be correct
- May have to be modified often as code changes quickly
- Can be written by coders, QA team needs not be involved in the construction of the model

8 Conclusion

A lot of tools are required to automatically create a model from existing test cases: a translator for the language and framework the test suite in question is written in, a grammar that relates commands to state-changes in the tests, and a test adapter that can abstract (5.2.3) test values.

Unfortunately, several of them, and most importantly the interpretation component, have to be application specific. This means that for the model construction, and later on the test generation, to work for any given application, at least three components have to be written first. These components are quite complex by themselves, and as such, constructing the model manually may very well be cheaper than doing everything that is required to do it automatically. Projects with very large pre-existing test suites might be an exception however, since the effort required to write models for the entire application might be monumental.

Thus, in conclusion, automatic generation of software models from test suites is possible, but requires a great deal of initial effort to give the modeller the necessary application-specific information.

References

[Ammann et al., 2002] Ammann, P., Black, P. E., and Ding, W. (2002). Model checkers in software testing. Technical report, NIST-IR 6777, National Institute of Standards and Technology. Also available at `http://hissa.nist.gov/~black/Papers/ir6777.pdf`.

[de Moura et al., 2003] de Moura, L., Owre, S., and Shankar, N. (2003). The SAL language manual. Technical report, SRI International. Also available at `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.87.8303&rep=rep1&type=pdf`.

[Jääskeläinen et al., 2009] Jääskeläinen, A., Kervinen, A., Katara, M., Valmari, A., and Virtanen, H. (2009). Synthesizing test models from test cases. In Chockler, H. and Hu, A. J., editors, *Proceedings of the 4th International Haifa Verification Conference on Hardware and Software, Verification and Testing (HVC 2008)*, volume 5394 of *Lecture Notes in Computer Science*, pages 179–193, Berlin, Heidelberg. Springer. Also available at `http://www.cs.tut.fi/~clark/doc/HVC'08-preliminary.pdf`.

[Utting et al., 2006] Utting, M., Pretschner, A., and Legeard, B. (2006). A taxonomy of model-based testing. Technical report. Also available at `http://www.cs.waikato.ac.nz/pubs/wp/2006/uow-cs-wp-2006-04.pdf`.

[Voronkov, 2010] Voronkov, A. (2010). Linear temporal logic. In *Logic and Modeling draft*, chapter 14. Also available at `http://www.voronkov.com/lm_doc.cgi?what=chapter&n=14`.

Appendices

A Model code

A.1 Touch

```
1  touch: CONTEXT =
2  BEGIN
3      FileType : Type = {void, file, voidpath};
4      Action   : Type = {create, cthrough, modify, none};
5      Error    : Type = {no, nosuchdir, nopermission};
6
7  touch: MODULE =
8  BEGIN
9      INPUT canWrite   : boolean
10     INPUT canRead    : boolean
11     INPUT ownsFile   : boolean
12     INPUT ownsDir    : boolean
13     INPUT target     : FileType
14     INPUT nocreate   : boolean
15     INPUT dangling   : boolean
16     INPUT isFIFO     : boolean
17     INPUT isEmpty    : boolean
18     INPUT isDir      : boolean
19
20     LOCAL touchedDangling : boolean
21     LOCAL touchedDir : boolean
22     LOCAL touchedEmpty : boolean
23     LOCAL touchedVoidPath : boolean
24     LOCAL touchedFIFO : boolean
25     LOCAL noCreateNoFail : boolean
26     LOCAL touchedUntouchable : boolean
27     LOCAL touchedOthers : boolean
28     LOCAL touchedReadOnly : boolean
29
30     OUTPUT error     : Error
31     OUTPUT action    : Action
32
33   INITIALIZATION
34
35     error = no;
36     action = none;
37
38   TRANSITION [
39     (target = file AND dangling = true AND (ownsFile = true OR canWrite =
           true) AND nocreate = false)
40         -->
41         error' = no;
42         action' = cthrough;
43         touchedDangling' = true;
44    []
45     (target = voidpath)
46         -->
47         error' = nosuchdir;
48         action' = none;
49         touchedVoidPath' = true;
```

```
50      []
51       (target = void AND nocreate = true AND (canWrite = true OR ownsDir =
             true))
52           -->
53           error' = no;
54           action' = none;
55           noCreateNoFail' = true;
56      []
57       (target = void AND (canWrite = true OR ownsDir = true) AND nocreate =
             false)
58           -->
59           error' = no;
60           action' = create;
61      []
62       (target = file AND canWrite = false AND ownsFile = false)
63           -->
64           error' = nopermission;
65           action' = none;
66           touchedOthers' = true;
67      []
68       (target = file AND dangling = false AND (ownsFile = true OR canWrite =
             true))
69           -->
70           error' = no;
71           action' = modify;
72           touchedDir' = isDir;
73           touchedEmpty' = isEmpty;
74           touchedFIFO' = isFIFO;
75           touchedUntouchable' = (canWrite = false AND canRead = false);
76           touchedReadOnly' = (canRead = false);
77    ]
78  END;
79
80  END
```

A.2 WiFi driver

```
1   wifi: CONTEXT =
2   BEGIN
3       Operation : Type = {scan, show, connect, remoteUp, remoteConnect,
            remoteDisconnect, disconnect, traffic, suspend, resume};
4       Network   : Type = {none, this, remote};
5
6       wifi: MODULE =
7       BEGIN
8           %% State
9           LOCAL suspended : boolean
10          LOCAL up : Network
11          LOCAL connected : boolean
12          LOCAL remoteConnected : boolean
13
14          %% Goals
15          LOCAL goalTrafficAfterResume : boolean
16          LOCAL goalTraffic : boolean
17          LOCAL goalResume : boolean
18          LOCAL goalReconnect : boolean
19          LOCAL goalDisconnect : boolean
20          LOCAL goalConnect : boolean
```

```
21          LOCAL goalConnectRemote : boolean
22          LOCAL goalShow : boolean
23          LOCAL goalScan : boolean
24          LOCAL goalCreate : boolean
25
26          %% Input
27          INPUT op : Operation
28
29          %% Output
30          OUTPUT network : Network
31
32          INITIALIZATION
33          suspended = false;
34          up = none;
35          connected = false;
36          remoteConnected = false;
37
38          goalTrafficAfterResume = false;
39          goalTraffic = false;
40          goalResume = false;
41          goalReconnect = false;
42          goalDisconnect = false;
43          goalConnect = false;
44          goalConnectRemote = false;
45          goalShow = false;
46          goalScan = false;
47          goalCreate = false;
48
49          TRANSITION [
50              %% Connect to networks
51              (op = connect AND NOT (up = none) AND suspended = false AND
                    connected = false)
52              --> connected' = true;
53              goalConnect' = true;
54              goalConnectRemote' = (up = remote);
55              goalReconnect' = goalDisconnect AND up = remote;
56            []
57              (op = connect AND up = none AND suspended = false AND connected
                     = false)
58              --> up' = this; connected' = true;
59              goalConnect' = true;
60              goalCreate' = true;
61            []
62              %% Disconnect
63              (op = disconnect AND connected = true AND remoteConnected =
                    true AND suspended = false)
64              --> connected' = false; up' = remote;
65              goalDisconnect' = true;
66            []
67              (op = disconnect AND connected = true AND remoteConnected =
                    false AND suspended = false)
68              --> connected' = false; up' = none;
69              goalDisconnect' = true;
70            []
71              %% Remote connections
72              (op = remoteConnect AND NOT (up = none) AND remoteConnected =
                    false)
73              --> remoteConnected' = true;
```

```
74              []
75                (op = remoteDisconnect AND remoteConnected = true AND connected
                      = true)
76                --> remoteConnected' = false; up' = this;
77              []
78                (op = remoteDisconnect AND remoteConnected = true AND connected
                      = false)
79                --> remoteConnected' = false; up' = none;
80              []
81                (op = remoteConnect AND up = none)
82                --> up' = remote; remoteConnected' = true; goalDisconnect' =
                     false;
83              []
84                %% Show network status
85                (op = show AND connected = true AND suspended = false)
86                --> network' = up;
87                goalShow' = true;
88              []
89                (op = show AND connected = false AND suspended = false)
90                --> network' = none;
91              []
92                %% Scan
93                (op = scan AND suspended = false)
94                --> network' = up;
95                goalScan' = true;
96              []
97                %% Traffic
98                (op = traffic AND NOT (up = none) AND connected = true AND
                     remoteConnected = true AND suspended = false)
99                -->
100               goalTraffic' = true;
101               goalTrafficAfterResume' = goalResume;
102             []
103               %% Suspend/resume
104               (op = suspend AND suspended = false)
105               --> suspended' = true;
106             []
107               (op = resume AND suspended = true)
108               --> suspended' = false;
109               goalResume' = true;
110         ]
111
112     END;
113
114     no_connect_without_up : THEOREM  wifi |- G(connected = true => NOT (up
          = none));
115 END
```

A.3 tbase.c

```
1  tbase: CONTEXT =
2  BEGIN
3      Operation : Type = {busadd, register, unregister, getdrv, putdrv,
           register_firmware, register_class, register_sysdev};
4      Change : Type = {up, down, equal};
5
6      tbase : MODULE =
7      BEGIN
```

```
8           %% State
9           LOCAL deviceRegistered : boolean
10          LOCAL driverRegistered : boolean
11          LOCAL firmwareRegistered : boolean
12          LOCAL classRegistered : boolean
13          LOCAL sysdevRegistered : boolean
14
15          %% Goals
16
17          %% Input
18          INPUT op : Operation
19
20          %% Output
21          OUTPUT refcount : Change
22
23          INITIALIZATION
24          deviceRegistered = false;
25          driverRegistered = false;
26          firmwareRegistered = false;
27          classRegistered = false;
28          refcount = equal;
29
30          TRANSITION [
31              (op = busadd)
32              --> deviceRegistered' = true;
33            []
34              (op = register AND deviceRegistered = true)
35              --> driverRegistered' = true;
36            []
37              (op = unregister)
38              --> driverRegistered' = false;
39            []
40              (op = getdrv)
41              --> refcount' = up;
42            []
43              (op = putdrv)
44              --> refcount' = down;
45            []
46              (op = register_firmware)
47              --> firmwareRegistered' = true;
48            []
49              (op = register_class)
50              --> classRegistered' = true;
51            []
52              (op = register_sysdev)
53              --> sysdevRegistered' = true;
54          ]
55
56          END;
57  END
```

A.4 svc.startd

```
1   startd: CONTEXT =
2   BEGIN
3       Operation : Type = {import, disable, enable, oprefresh, coredump, exited,
            signal};
4       State : Type = {void, online, disabled, maintenance};
```

```
5       Command : Type = {start,stop,refresh,noop};
6
7       %% Functions
8       command(newState : State): Command = IF newState = disabled THEN stop
            ELSE IF newState = online THEN start ELSE noop ENDIF ENDIF;
9
10      startd : MODULE =
11      BEGIN
12          %% State
13          LOCAL is : State
14          LOCAL onlyRetryEnable : boolean
15          LOCAL onlyRetryDisable : boolean
16          LOCAL onlyRetryRefresh : boolean
17          LOCAL onlyRetry : boolean
18
19          %% Goals
20          LOCAL goalImportEnabled : boolean
21          LOCAL goalImportDisabled : boolean
22          LOCAL goalDisable : boolean
23          LOCAL goalEnable : boolean
24          LOCAL goalMaintenanceFromDisable : boolean
25          LOCAL goalMaintenanceFromEnable : boolean
26          LOCAL goalMaintenanceFromRefresh : boolean
27          LOCAL goalDisableFailThenSucceed : boolean
28          LOCAL goalEnableFailThenSucceed : boolean
29          LOCAL goalRefreshFailThenSucceed : boolean
30          LOCAL goalCoreDump : boolean
31          LOCAL goalExited : boolean
32          LOCAL goalSignal : boolean
33          LOCAL goalDisableNotDefined : boolean
34          LOCAL goalTimeout : boolean
35          LOCAL goalDoRefresh : boolean
36
37          %% Inputs
38          INPUT op : Operation
39          INPUT initialState : State
40          INPUT commandSucceeds : boolean
41          INPUT stopExists : boolean
42          INPUT commandTimeout : boolean
43
44          %% Outputs
45          OUTPUT checkCommandRun : Command
46
47          DEFINITION
48          onlyRetry = onlyRetryEnable OR onlyRetryDisable OR onlyRetryRefresh
49
50          INITIALIZATION
51          is = void;
52          goalImportEnabled = false;
53          goalImportDisabled = false;
54          goalDisable = false;
55          goalEnable = false;
56          goalMaintenanceFromDisable = false;
57          goalMaintenanceFromEnable = false;
58          goalMaintenanceFromRefresh = false;
59          goalDisableFailThenSucceed = false;
60          goalEnableFailThenSucceed = false;
61          goalRefreshFailThenSucceed = false;
```

```
62          goalCoreDump = false;
63          goalExited = false;
64          goalSignal = false;
65          goalDisableNotDefined = false;
66          goalTimeout = false;
67
68          TRANSITION [
69              %% Here be dragons
70              %% Note that the test adapter should enumerate where necessary
71              %% It should also ALWAYS check that the state reported by the
                    model is equivalent to the state of the application
72              (op = import AND is = void AND NOT(initialState = void) AND NOT
                    (onlyRetry = true))
73              --> is' = initialState;
74              checkCommandRun' = command(initialState);
75              goalImportEnabled' = IF initialState = online THEN true ELSE
                    false ENDIF;
76              goalImportDisabled' = IF initialState = disabled THEN true ELSE
                     false ENDIF;
77            []
78              (op = disable AND NOT(is = disabled OR is = void) AND
                    commandSucceeds = true AND onlyRetry = false AND stopExists
                    = true AND commandTimeout = false)
79              --> is' = disabled;
80              checkCommandRun' = stop;
81              goalDisable' = true;
82            []
83              (op = disable AND NOT(is = disabled OR is = void) AND
                    commandSucceeds = false AND onlyRetry = false AND stopExists
                     = true AND commandTimeout = false)
84              % If the test run indicates a fail, the test adapter must cause
                     the program to fail
85              % It must also check that stop was not execute several times as
                     indicated by #006
86              % depending on whether the commandSucceeds was error or
                    error_maintenance
87              --> onlyRetryDisable' = true;
88            []
89              (op = disable AND NOT(is = disabled OR is = void) AND
                    commandSucceeds = false AND onlyRetryDisable = true AND
                    stopExists = true AND commandTimeout = false)
90              --> is' = maintenance;
91              goalMaintenanceFromDisable' = true;
92            []
93              (op = disable AND NOT(is = disabled OR is = void) AND
                    commandSucceeds = true AND onlyRetryDisable = true AND
                    stopExists = true AND commandTimeout = false)
94              --> is' = disabled;
95              checkCommandRun' = stop;
96              goalDisableFailThenSucceed' = true;
97              onlyRetryDisable' = false;
98            []
99              (op = disable AND NOT(is = disabled OR is = void) AND
                    stopExists = false AND onlyRetry = false AND commandTimeout
                    = false)
100             % Enumerate: 33,34
101             --> is' = maintenance;
102             goalDisableNotDefined' = true;
```

```
103             onlyRetryDisable' = false;
104           []
105             (op = enable AND NOT(is = online OR is = void) AND
                    commandSucceeds = true AND onlyRetry = false AND
                    commandTimeout = false)
106             --> is' = online;
107             checkCommandRun' = start;
108             goalEnable' = true;
109           []
110             (op = enable AND NOT(is = online OR is = void) AND
                    commandSucceeds = false AND onlyRetry = false AND
                    commandTimeout = false)
111             % Enumerate: 11,43
112             --> onlyRetryEnable' = true;
113           []
114             (op = enable AND NOT(is = online OR is = void) AND
                    commandSucceeds = false AND onlyRetryEnable = true AND
                    commandTimeout = false)
115             --> is' = maintenance;
116             goalMaintenanceFromEnable' = true;
117             onlyRetryEnable' = false;
118           []
119             (op = enable AND NOT(is = online OR is = void) AND
                    commandSucceeds = true AND onlyRetryEnable = true AND
                    commandTimeout = false)
120             --> is' = online;
121             checkCommandRun' = start;
122             goalEnableFailThenSucceed' = true;
123             onlyRetryEnable' = false;
124           []
125             (op = oprefresh AND is = online AND commandSucceeds = true AND
                    onlyRetry = false AND commandTimeout = false)
126             --> is' = online;
127             checkCommandRun' = noop;
128             goalDoRefresh' = true;
129           []
130             (op = oprefresh AND is = online AND commandSucceeds = false AND
                     onlyRetry = false AND commandTimeout = false)
131             --> onlyRetryRefresh' = true;
132           []
133             (op = oprefresh AND is = online AND commandSucceeds = false AND
                     onlyRetryRefresh = true AND commandTimeout = false)
134             --> is' = maintenance;
135             goalMaintenanceFromRefresh' = true;
136             onlyRetryRefresh' = false;
137           []
138             (op = oprefresh AND is = online AND commandSucceeds = true AND
                    onlyRetryRefresh = true AND commandTimeout = false)
139             --> is' = online;
140             checkCommandRun' = noop;
141             goalRefreshFailThenSucceed' = true;
142             onlyRetryRefresh' = false;
143           []
144             (op = coredump AND commandSucceeds = true AND onlyRetry = false
                     AND NOT(is = void))
145             % Enumerate: 15,16,17,18,23
146             % Test cases do not indicate the behavior if restart fails..
147             --> is' = online;
```

```
148               checkCommandRun' = start;
149               goalCoreDump' = true;
150             []
151               (op = exited AND commandSucceeds = true AND is = online)
152               % All children exit as per test case #19
153               % Online status should not change
154               % Enumerate: 19,20,21,22
155               % Again, test cases do not indicate what to do if restart fails
156               --> checkCommandRun' = start;
157               goalExited' = true;
158             []
159               (op = signal AND is = online AND commandSucceeds = true)
160               % Again, test cases do not indicate what to do if restart fails
161               % Enumerate: 24,25,26,27,28,29,30
162               % Enumerate signals: fatal, hwerr
163               --> checkCommandRun' = start;
164               goalSignal' = true;
165             []
166               (onlyRetry = false AND NOT(is = void) AND (
167               (is = online AND commandTimeout = true AND (op = disable OR op
                    = oprefresh))
168               OR
169               (NOT(is = online) AND commandTimeout = true AND op = enable)
170               ))
171               --> is' = maintenance;
172               goalTimeout' = true;
173           ]
174     END;
175 END
```

B Notes

The following sections contain the notes I made during the development of each model. The only modifications are the removal of timestamps, and general formatting.

B.1 touch

I've now gone through all the test cases for the command line tool "touch", and here are my initial observations: - Knowledge about the internal implementation of the tool is not necessary - Comments in the test cases are extremely valuable when developing the model - Every test case can initially be "translated" to one or more state-transitions - These state-transitions become very specific, as each test case only tests a very limited functionality of the application. Many test cases have overlapping transitions

From what I can tell, there needs to be at least two phases:

1. Abstraction of test cases to state-transitions

2. Further abstraction to make state-transitions more general (less restrictive) by making the "rules" inherently present in test cases into theorems instead

This seems to lead to a clean set of states and transitions that quite accurately represent the functionality of the program where the theorems capture the essence of the motivation between the test cases.

Also, the fact that this utility does not keep state between invocations, and every operation therefore can be regarded as atomic, all the theorems are quite straightforward to write as they do not require globally/finally.

I'm starting to think that writing theorems is not really necessary for what we're trying to do. Instead, we should be writing reachability goals so that we can use sal-atg to generate test cases. I have also observed that many of the test cases are there due to odd behavior in the program, but very few test that the program works as it should in "normal" situations.

Furthermore, some test cases are hard to model due to their boundless nature. For instance those that involve checking numerical operations (touch –ref f –date="-5 days" f). These will not gain much from being modeled because they have an infinite amount of combinations, and thus cannot be checked exhaustively.

I've now finished the model for the command line utility, and from what I can tell, it will indeed be able to generate more tests than are currently in the suite. More specifically, it will complete the suite in the sense of testing everything that can be inferred from the test cases, such as negations of every test case. For instance, the generated suite will contain a test for the case when a user has read access to a file, but no write access and no ownership, which the current suite does not have.

I am not convinced that SAL is the best modelling tool, however, since it has two major flaws: 1. sal-atg does not, by default print the values of the outputs of each step. This can be enabled with the –fullpath option, but the feature is bugged as it shows any OUTPUT values only with the first value they are given 2. sal-atg does not lend itself easily to writing an adapter that can take the generated test runs and execute the corresponding commands. That said, the syntax is quite straightforward, and is (in my opinion) cleaner than for instance PyModel. Considering we at this stage do not need to interface the test runs with any other program, SAL should be fine for now. I will

however be looking into writing an emacs syntax file for SAL to make editing easier (and prettier).

This last point about needing an adapter is quite vital with regards to the prospect of automating the modelling from test suites. Assuming that one can parse comments and commands from the test suites, one still has the problem of being able to generate the executable code from the models generated test runs. This will apply no matter what modelling framework being used - an adapter will have to be written (i.e. it probably cannot be generated).

Metrics:

- model size: 80 LOC
- man-hours: 5
- transitions: 6

B.2 WiFi driver

Starting modeling of WiFi driver from SCT test suite @ 10:30 AM, Jan 28th

I can tell already that this is going to be tough to get right for a computer.. The biggest problem I'm seeing now is that the tests all seem to run tools that give output with regards to the current state of the device, and then compare that output to the data given in the setup stage of the test script. Thus, the test cases actually test the tools more so than the driver. The reason I think this might be a problem for a computer to do is that it would have to "understand" what lies behind the tools, and that they are only views into the state of the device. For instance, the two test cases tp_..._004.ksh and tp_..._005.ksh test essentially the same thing - does the device driver successfully send and receive packets, but does so using two different tools (ping and runuperf). A model synthesizer would need the ability to "understand" this, and merge them into a single aspect of the model (i.e. a traffic goal). Trying out both these types of traffic should be left up to the adapter, as it the differentiation does not exist in the model (this is discussed further below).

Another issue I'm detecting is that when considering the test cases one by one, and building the model as one goes through each one, the model changes rapidly in the beginning since the test cases only reveal small aspects of the program at the time. This causes a lot of unnecessary work as the model is rewritten time and time again through the first couple of test cases to accomodate the new features exposed by the test cases. It seems to me as though considering all the test cases first, and then building the model, might prove a more efficient approach.

Note that this approach is only more efficient if one manages to reason about the test cases globally and deduce the essence of the programs behavior. The aim of the model is to express how the program is meant to work, or rather, how it is meant to behave. The role of the test cases on the other hand is usually to test that different specific usage scenarios work correctly. The main problem in doing model abstraction from test cases automatically is to make the computer able to "understand" the purpose of a program based solely on sequential invocations of that program and expected outputs in each step. This is tricky, even for a human.

Furthermore, I'm noticing that the model does lose some implementation specific data due to the abstractions. For instance, the test suite has test cases for both sending pings

and "normal" data between hosts in an Ad-Hoc network, whereas in the model, this is simply modeled as a traffic "operation" that succeeds if the network is up and both clients are connected. In fact, a proper model would probably not have this in at all, since it could be assumed that if an interface is connected, traffic should be working. The fact that in some cases, traffic can work, but not ping, is an implementation specific problem that the test case was written for. Due to the abstraction, the model might not catch these kind of problems. This does of course depend on the adapter used to convert generated test runs to test cases, which might include the distinction between these two types of traffic.

Seems as though the model is nearly done already...
Device drivers seem to be easier to model than command line utilities due to their inherenly stateful nature. One thing to note about the generated test run is that it contains unnecessary steps, such as multiple show() calls in a row or multiple alternating suspend/resume calls. These can be removed by the adapter if it can be made aware of what operations are "passive", and which modify the state of the system. It would also need to know (in order to remove such redundancy), which calls "undo" each other, such as suspend and resume, and only run such cycles once.

As for the usefulness of this model, it is already apparent by looking at the generated test run, that some aspects of the driver that weren't previously tested now are. For instance, scans are now performed after suspend/resume as well, something which was not part of the initial test suite.

I have also finished two emacs major modes, one for SAL files providing both indenting and highlighting, and one providing highlighting for the output of sal-atg.

No, will write a file that outlines the basic structure of the model, as well as how the model incorporates the different test cases (see Model Description below)

Metrics:

- model size: 103 LOC
- man-hours: 3(!)
- transitions: 14

B.2.1 Model description

This model is based on a subset of the test suite for wireless device drivers which can be found in driver-wifi/tests. The included tests deal only with Ad-Hoc mode and simple power management (suspend-resume). Note: In all the test files, any reference to ibss means Ad-Hoc mode.

Essentially, the ibss-related test cases all deal with various Ad-Hoc setup scenarios. More specifically:

ibss/..1

- Creates an Ad-Hoc network, and then gets a remote host to connect to it.
- Then sets up an Ad-Hoc network on the remote host and connects the current machine to it.
- This is performed with several different security modes, but this is not in the model since it should apply to all test cases, and thus can be put in the adapter.

In the model, the currently active network is in the state variable "up", which is either "none" (no network up), "this" (local computer started network) or "remote" (remote computer started network). Whether we or another computer are currently connected are stored in the "connected" and "remoteConnected" state variables respectively.

ibss/..2

- Tests that, after connecting to a network, the utility for showing information about the current connection shows the same information as the network was set up with.
- Also checks that a scan will reveal a network with the appropriate network settings when an Ad-Hoc network is set up on a remote host.
- The checks are then performed with remote and local machine reversed, and with all different security modes.
- The security modes can again be put in the adaptor, since the model behaviour should be independent of the security mode.

The model contains two operations, "show" and "scan", that represent calling the utilities for performing these two actions. These are "passive" operations in that they don't change the state of the program (except their respective goal variables to ensure that they both get called at least once) The two goals "goalShow" and "goalScan" ensure that we test both these utilities.

ibss/..3

- Creates an Ad-Hoc network, connects a remote host, disconnects from the network, and then attempts to reconnect since an Ad-Hoc network should remain open even if the original host goes down.
- Again, this is also tested with all security modes, but that can be ignored in the model.

This is tested by the reachability of goalReconnect, which is only reached if we have recently been disconnected, and the currently active network is hosted by the remote party.

ibss/..4

- Tests whether two nodes that are both connected through the Ad-Hoc network can ping each other

ibss/..5

- Tests whether two connected nodes can send data to each other

ibss/..6

- Tests whether NFS disturbs data travelling between two nodes

In the model, these three are abstracted away as a "traffic" command. It is up to the adapter to try different types of traffic. The only thing we do in the model is set up a goal that at some point traffic should be sent, and the restriction that traffic can only be sent when both parties are connected.

B.3 NWAM

Determined server/daemon application to be the NWAM daemon. Have been looking through the test suite, and the syntax of some of the commands are pretty confusing.. Will have to dig around in the libraries for a while to get a grasp on how it is working.

Have been starting to look through the documentation of NWAM to understand the concepts behind it to better understand the test cases. Info mostly from the Oracle NWAM support website. Some of the data on the Korn Shell commands come from the IBM Boulder info center article on Korn. Also, chatted to Daniel about his project, and determined it's not very similar, although quite interesting.

After walking through the documentation mentioned above, as well as reading through a bit of the testing library code in src/lib/ksh/*.ksh, I am now starting to get a grasp of how the system works. NWAM has four different concepts that are important to grasp:

- NCP: Network Configuration Profile - A set of NCUs. only one NCP may be enabled at any given time
- NCU: Network Configuration Unit - A configuration for a single link or interface (Ethernet port, WLAN, VLAN, Tunnel, VNIC, etc..)
- Location: Network environment configuration with optional enabling conditions. For instance, a location might only be enabled if a certain NCP is active. Configures DNS, hostname, IP filtering, etc.. Only one may be active at any given time
- ENM: External Network Modifier - External applications that should be launched when certain conditions are satisfied, and that should be shut down when the conditions are no longer satisfied. Many may be active simultaneously. Next, for the commands in the test cases:
- startup creates variables that are in use throughout the testing process, as well as create a default NCP, NCU, Location and ENM (defaulting to enabled)
- nwt_install <type:name> creates a new NCP, NCU, Location or ENM
- nwt_enable <type:name> enables the given NCP, NCU, Location or ENM
- nwt_disable <type:name> does the opposite of nwt_enable
- start_nwam starts the NWAM daemon
- nwt_verify <type:name> <state>... checks that the given NCP, NCU, Location or ENM is in all the states given in the state list (online, offline, applied, etc...)

Every NCP, NCU, Location and ENM have their configuration values in a dictionary named $(type)_$(name).
Common parameters set here are:

activation-mode One of conditional-all, conditional-any

conditions List of conditions to check, such as "ip-address is 10.0.0.2" or "ip-address is-in-range 10.0.0.0/8"

_name The name of the NCU's underlying interface

ipv4-addr IPv4 address of the NCU

Notice also that in the top of each test case, a variable called ITERATORS is defined. This is a list of values that should all be tried for the current test case. This means the calling test harness will try all values for those variables. For instance, with ITERATORS="MODE", another variable called MODE_VALS will be defined and contain a list of the values for MODE to try. The harness will then call the function once with every value of MODE_VALS in the $MODE variable.

NOTE: There is an error in one of the test cases: tp_wired_changed_by_ifconfig.ksh. The last nwt_verify line should verify offline, not online according to the strategy description.

This is harder than the previous elements to model, though I am not sure whether it is because of SAL, or because it is hard to model in general. The part I'm struggling with is how to represent the different NWAM object types in SAL (e.g. NCP, NCU, ENM, Location), and especially, how I should model the conditions between them (a Location might only be enabled if a certain NCU is online, etc...)

I'm starting to think that the main problem is that I am trying to model the installing of these objects as well, whereas this is really just a part of the test harness setup, and not a property of the NWMA daemon. Thus, all the model should really consider itself with is whether a certain interface is up or down, it shouldn't deal with new interfaces being added, or being removed. That is a part of the operating environment, not an input. If I take the adding/removing out of the equation, a couple of problems still remain:

1. How to represent conditional activation/deactivation of ENMs and locations? Perhaps this could be done with arrays of arrays? Does SAL support them?

2. How many NCUs, ENMs, etc. should the model deal with? Theoretically, it does not really matter, two should be enough based on the examples in the test cases..

Seems SAL does support multi-dimensional arrays through "Array INDEX_RANGE of Array INDEX_RANGE of TYPE" Maybe conditions could be modelled with the two indices object type (ncu,ncp,enm,loc) and object number. Each cell could then be one of three: ignore,online,offline. The problem with this approach of course is that online/-offline aren't the only conditions available.. Perhaps just use a boolean for the cells: true = depends, false = no condition, and then leave it up to the adapter to do the enumeration of possible dependencies? In fact, it turns out that we need a four-dimensional array, because we also need to store the "source" of the condition (i.e. which object has this condition)... There must be a better way of abstracting these dependencies away and let the adapter enumerate instead..

I've been trying to come up with a solution to this over and over again in my head over the past 30 minutes, but have not come up with any viable abstraction of the constraints in the model; they all end up essentially being a full implementation of the constraint system, or something so abstract that it will be impossible to write an adapter for it. I will have a chat to Paddy before I come in next week to see if he can help me in coming up with another way of modeling this. Until then, there is not much I can do - this is a blocker...

I've now had a meeting with Paddy, and the results weren't very promising. The problem with the application I'm currently modeling is that its behavior is very configuration dependent. Depending on the configuration, a given event can in some cases enable an interface, and in other cases disable it. In essence, the configuration defines the

behavior of the application, and thus, when modeling, we should really create one model per configuration. The problem with this, of course, is that there are an infinite possible configurations, making it impossible to create the needed models. Furthermore, even after creating a configuration-specific model, any test cases generated from that model would only be valid for that given configuration of the application. This is very much in contrast to what we would want to be able to do - which is to generate test cases for different "types" of configurations. But again, the configurations cannot be easily abstracted in this application since they specify dependencies that completely alter the behavior of the system.

At this point, we have a couple of choices as to how we want to continue:

1. Model the system without configuration options such as dependencies
2. Pick a different server/daemon application, and model that instead
3. Go back to modeling command-line utilities and drivers, and leave servers for another project

After a discussion with Lian, we have decided to abandon this server/daemon, and attempt to find another, more suitable, application to model.

B.4 tbase.c

I have been told by Lian that we are waiting to hear back from Solaris QA staff to get daemons with test cases that we can model, and that in the meantime, I should model another device driver instead.

Still looking for a suitable device driver with a test suite to model. Have a meeting with Cristina soon where I will discuss with her the plan for the remainder of this project considering we've already covered most of the primary work outlined in the Scoping Document. Been looking a bit at the tbase kernel driver from LTP (The Linux Test Project), but it seems as though all it does is execute a series of system calls and check their return value. There is not state..

Turns out I'm just a bit tired... Of course it makes sense for the test cases to only contain method calls and check their return value. We're not modeling the test suite, we're modeling the application that the test suite is testing!

Have now had meeting with Cristina, and we have determined that I will keep modeling more applications/drivers, and include these in my final report. Also, she told me it would be nice if the final report included some discussion on whether the model would be different if it was developed from scratch by a developer or designer that knew the specs or application source. I don't think it should be a problem to include this data in the final report.

Have started modeling the kernel driver for handling devices (tbase.c). The documentation of the test cases in tbase.c are all over the place, and quite confusing at times. I have also noticed that some operations have a no-op end result because they execute corresponding create and close operations (e.g. test_create_file runs driver_create_file, and then driver_remove_file just to test that running driver_create_file will actually work). The problem with this is that no program state has changed as a result of the operation, and the running of driver_create_file does not seem to change the state of the program, meaning it does not make sense to put this in the model. However, these operations should still be tested.

I'm starting to see a pattern here in that most programs have test cases that are either algorithm-specific corner cases or "will it run" tests that do not abstract easily into the model, meaning that you could not replace the test suite with a single (or multiple) models, but you will rather have to use both. That said, the "will it run" operations **can** be modeled by just creating a number of inputs that have no effect on the state apart from reaching a certain goal.

Model completed already... This was actually very straightforward, since the test suite did not really present much of what kind of state was being kept in the driver/kernel; it only ran calls that it expected to succeed, and reported test failure if they didn't succeed. The simplicity does come at a cost though, because the model is now so basic that it cannot generate any more test cases than already exist.. The only difference would be that it might try more interleaving of commands than the current test cases do.

B.5 svc.startd

Have now gone found the next application I will be modeling. It is svc.startd, the "Service Management Facility master restarter" for Solaris. The tool takes care of starting, stopping and restarting services that run on a box safely, and can even do so with dependency resolution and such. Due to the experience with nwam, I have chosen to only look at the test cases for the direct invocations of svc.startd through svcadm (tests/methods). Again, it will take some time to go over the test cases and understand the testing framework being used for this application's tests. Note that even just the test cases for direct invocations are fairly numerous and number $\tilde{6}500$ LOC, so this might take a while...

Again, I'm noticing that some subtle differences don't lend themselves well to abstraction. For instance, for both enable, disable and refresh, there is a differentiation in the test suite between the corresponding command's exit code:

- If it is listed as a straight-to-maintenance code, the service is set to be in maintenance mode
- Otherwise, with normal error messages, the command is executed multiple times until it either a) gives up and goes to maintenance mode or b) succeeds

This cannot be easily represented in the model unless we keep a counter as to how many times a command fails, and even then, it would depend on the configuration whether it should go to maintenance mode or not. What I'm currently thinking is to add a small integer that counts down every time a command fails, and then, if it reaches zero, send the process to maintenance mode. Furthermore, I'll set commandResult as an input, so it will a) be easy for the test adapter to know what kind of environment it should generate, and b) differentiate between exit code error and error_maintenance so we can send some straight to maintenance and some after a certain number of tries.

The problem with that approach of course is that by only decrementing the counter if we get commandResult error, that means the model won't "know" that it has to keep trying the same operation until it either succeeds or the counter reaches 0. This has to somehow be explicitly stated in *every* guard to prevent it from taking other transitions after an error. I think SAL has a way of specifying that after taking a certain transition, only some other transitions are available, but I am not sure yet. If that was the case, we would not need a counter at all, we could just model the transitions so that if commandResult = error, there are only two transitions available: one where commandResult

= success and one where commandResult = error (again). The first one sends the system back to where it can take any transition, the second causes the service to go into maintenance mode.

I've now emailed Paddy about this, and hopefully he'll give me a pointer as to how this might be done with SAL. It is of course possible by adding an additional guard to every single other transition, but then you would need one such "lock" variable for every transition that might contain retries.. Not pretty. Got to 14/44 today, will continue next week.

Unfortunately, Paddy knew of no way of doing this nicely in SAL apart from setting a guard on every other transitions. It's not pretty, but I suppose it is the way it will have to be. Have gone over the model now and guarded all transitions, so the model now fully captures the different behaviors exhibited by the original test cases (i.e. retry = fail, retry = success).

Having a bit of trouble with TC #33: How to go about modeling whether a certain program has a stop operation or not. Could have an input operation stop_noexist, but that would not really be correct since the system can still use operation stop, which should not be possible. Alternately, I could require the test adapter to enumerate this, but that might be difficult as well, because this is a behavior that should be present in the model. I could use non-determinism on the stop operation, but that would make it difficult to do the test generation later on because the test script would not know when which of the outcomes are correct behavior or when it is an error. Could add an additional input that says whether there is a stop operation defined, and add that as a guard for all stop operations to determine the correct behavior. That is probably the best approach, because it allows simple test generation as well. The problem with this approach is that the input can still vary between each operation (stop could be defined in one run, but not in another..).

Now on TC #37. Not entirely sure how to model this considering we don't actually model multiple processes, and thus cannot dispatch signals to children. Skipping for now. Also, with regards to #38, I am not entirely sure about the distinction between the start and enable operations, so not sure how this will fit into the model. Similarly with #39.

Seems as though there are many of these additional operations (#38,#39,#40,#41). There also seems to be some overlap with previous test cases (#42 vs #4). Furthermore, the model has no notion of transient services as mentioned in #44. This is especially because the model is abstractly tied to only a single service, which is either transient or it isn't. If we wanted to model different "modes" applications can be in, these all have different behavior, and should thus be placed in separate models.

Have now generated the tests for the model. Will look through the generated test runs after lunch.

I have been walking through the test cases, and one of the first thing I noticed is that there are some behaviors that the test cases do not specify, and thus that are not handled. An example of this is what should happen if the initial state for a service is "maintenance" and "import" is called. Should an error be generated? Should it indeed be placed in maintenance mode? Anything that is not explicitly stated in the test cases will be considered correct behavior. If the behavior is not defined, then anyone who runs the generated tests is sure to get test cases that exercise behaviors they haven't considered, hopefully leading to the writing of more explicit or elaborate test cases, which will again improve the model. The same can be found (as indicated with comments in the model) with operations such as exited and signal, where their suboperation (restart) fail.

Another thing I'm noticing is that the test adapter needs to be able to prioritize inputs. For instance, if commandSucceeds = true AND commandTimeout = true, then the path of commandSucceeds can be ignored. Same with op = disabled, commandSucceeds and stopExists.

There seems to be something strange about the output of sal-atg... It occasionally doesn't display variables, even though they change, and sometimes goals that are hit are ignored. Not sure what the cause of this is.. Unfortunately, it does make it a bit hard to work with the output. As to the usefulness of the test runs generated by the model, there are a couple of things worth noting:

- Their main difference from the existing test cases is that they test different combinations of inputs that might not have been considered in the test suite
- Second, the generated test cases are not very useful on their own, especially because of the previous point. The interesting part is the errors you would get when running them, that might indicate scenarios you do not have test cases for.
- The model is only as useful as the overall coverage of the test suite. Furthermore, it can only test the features of the system that are expose in the test suite.
- And finally, the test adapter (both ways) needs to be fairly sophisticated, especially because it should be able to do enumeration of environments that might cause certain inputs.

This leads me to thinking that doing automatic model-based test generation from test cases will:

1. never replace normal tests
2. not be possible to do completely automatically, since the test adapter will have to be written by hand
3. not be worthwhile, since it will probably be easier to write the model from scratch manually than to attempt to craft the test case abstracter + adapter.

This model is essentially finished, so I will be starting my report + preparation of my presentation for next week.