

Two Approaches to Automatic Testing of Operating Systems Kernels

Percy Pari-Salas
Padmanabhan Krishnan
Center for Software Assurance
School of Information Technology, Bond University
Gold Coast, QLD 4229 Australia
Email: {pparisal,pkrishna}@staff.bond.edu.au

Abstract

As operating system kernels are complex they are difficult to test. The difficulty in testing has traditionally been addressed by developing test suites. However test suites do not always provide information on the test purpose and design. They also lack flexibility when different scenarios need to be tested. Model-based testing can be used to capture the design as well as provide flexibility. However it can be expensive to adopt. In this paper we show how a model-based approach that reuses important parts of existing frameworks can be adopted. We use the scripts developed to test the HBA storage driver for the OpenSolaris operating system in the context of models to illustrate the benefits of model-based testing. We use SmartMBT to control and execute the actual tests.

1. Introduction

Operating system kernels are complex and difficult to test software. They usually include code developed by several different programmers, in different places and time. In such environment, it is hard to test the various functionalities and quality properties of the code. There are two main types of testing strategies, viz., black box and white box testing. White box testing focus on the actual code of the system and is usually done when issues such as coverage need to be addressed. Black

box testing on the other hand focuses on the functionalities of the system under test rather than the internal code. In the context of operating systems, black box testing does not require deep knowledge of the kernel internals, but the knowledge of the appropriate kernel module. This means that black box tests are portable between different implementations or similarly functionality of different modules. As the testing is mainly exercise the high level functionality black box tests are usually written using shell commands.

Consider, as an example, the testing of the HBA storage driver of the OpenSolaris operating system. This driver manages the storage devices and the file systems one can create on the devices. The interface will consist of commands such as *mount*, *umount*, *format* and *fdisk*. Black box tests for the implementation of this driver will contain scripts that invoke these commands exercising the different options available for each command. Additionally, the tests will also verify that there is a storage device connected and available. These scripts may require other scripts that check and get information from the environment, e.g. processor architecture (x86, 64-bits), size of the storage device, etc.

Black box testing, in general, has been implemented by developing *test suites*. Test suites are software packages (usually these scripts that perform shell invocations of the operating system commands) written for the express purpose of testing. They usually cover a wide range of functions and they are written to expose flaws that are likely

to be in the system.

There exist a range of test suites each designed for a specific purpose. These test suites are commonly used in certification where the functionality of the system is tested. For example, the Open Group¹ provides test suites to certify conformance with Unix and Posix standards. Test suites are also used in open communities as a way to ensure that the code contributed by different programmers maintain standard interfaces and do not break other functionalities. For example, the OpenSolaris community has an already developed test suite that aims to test the implementation of the storage drivers discussed earlier.

Many test suites, such the ones used for certification of conformance to standards are mature and provide adequate documentation. However in some other cases the main test design and purpose (i.e., what it is exactly being tested by a test suite) are not well documented. Additionally, in most operating systems it is difficult to define the boundaries between different “units of operation”. This means, functionalities at the OS level are highly integrated and an important number of interactions between different components need to be tested. It is not clear how existing test suites can be extended to test scenarios not initially considered.

In this paper we show model-based testing can capture both test design and purpose as well as be used to test new scenarios. The use of models to capture test design is well known [6, 7]. The models developed for testing can also be used to derive tests that guarantee certain type of coverage over the models. This can be combined to with other aspects such as random and interactive or exploratory testing where new scenarios can be explored dynamically. For a more detailed description of model-based testing and supporting tools we refer to [7].

An issue that is a hurdle in the adoption of model-based techniques is related to the cost of testing. This is especially true if scripts already developed cannot be reused. In this paper we show how a model-based approach can be set up so that

most, if not all, of the previous developed scripts can be reused. Thus we can capitalise on the benefits of model-based approaches without rewriting the entire testing framework. Note that the cost of developing the models is still an issue.

In the next section we present a brief overview of previous and related work. Then, in Section 3 we discuss briefly the general architecture of a model-based testing approach and how this architecture can be linked to the test suites approach used in testing operating systems. In Sections 4 and 5 we present the use of the framework in testing the OpenSolaris HBA storage driver. In this we show how the existing scripts are reused without change as well as how superficial changes to the structure of the scripts can lead to more extensive testing. Finally, in Section 6 we discuss our results.

2. Related work

The importance of making operating systems reliable has always been recognised but recently has attracted even more attention [5]. One of the ways of assuring reliability in which research has focused is formal verification. Formal verification can be impractical or even impossible for large systems. For example, systems whose size exceeds 10,000 lines of code are very difficult to verify [3]. However, Klein [3] presents an overview of projects in the area of verification of operating systems that implement microkernels (whose size is around 10,000 loc). Tanenbaum et al. [5] and Chou et al. [1] agree in that the main source of bugs in operating systems is the code of the drivers. Even in the case of microkernels, there is a special interest in insulate the kernel itself from the drivers to avoid that bugs in the later ones crash the entire system. The practical part of our work has focused on testing storage drivers. Sen [4] presented an approach called *concolic testing* which combines random exploration and symbolic execution and applied it to test the scheduler of a real-time operating system. Although, in this paper we focus on the modelling and reuse of scripts, our approach shares the principles of Sen’s approach. The main difference is that Sen’s

¹<http://www.opengroup.org/testing/>

applies his approach to source code while we work at the modelling level. Our approach also shares principles with Kandl’s work on automated verification and testing of embedded systems [2]. However, Kandl derives an automaton model from the source code and needs to define specific properties of the code to generate test cases. We build the models from existing test suites and modify the purpose of the tests dynamically. As we do not generate the models from the code, our approach is more suited for independent testing or assurance.

3. General Architecture

In a model-based testing approach tests are generated from a model of the system’s behaviour. Models are also used to specify the tests we intend to perform, i.e., to capture the test design. *Labelled transition systems* are perhaps the most widespread formalism used to model system’s behaviour. In this paper we use labelled transition systems.

Briefly, a labelled transition system is an abstract machine composed of a set of states, a set of actions and a transition relation that defines how the machine transits from one state to another. Each action has a name (or label) and the machine changes its state by executing an action. For a more comprehensive, general interest description of the use of labelled transition systems in testing we recommend [6].

Although the central element of the model-based approach is the system’s model, its advantage is the degree of automation that can be achieved. Thus, elements such as the testing tool and the proper system under test (SUT) are also important part of the testing framework. In Figure 1 we show a general architecture for the model-based testing approach.

Figure 1 shows the testing process driven by the system’s model and performed by the testing tool. The testing tool links itself to the SUT to verify its conformance to the model. The link between the testing tool and the SUT requires a middle layer in the communication channel, called the *Adaptor*. The Adaptor implements the *action word ap-*

proach. In this approach, each action (or transition label) in the model exercises a functionality in the SUT. Thus, the Adaptor, is usually implemented as a set of scripts which translates action names from the model into actual procedures or functions in the SUT. Moreover, the scripts used to implement an Adaptor must provide return values which can be used to determine the state of SUT and compare it with the expected state in the model. So each time an action is executed the tool listens the communication channel waiting for a “pass” or “fail” verdict.

The described architecture can integrate current test suites for operating systems in two different ways. The first one is to consider the current test suite (scripts) as functional units of the SUT. Thus, each action in the model maps to an specific script in the test suite. This approach helps to document the test suite and reuses its scripts without any change. It has another advantage in the fact that scripts can be exercised in different orders to the one specified into the test suite. This effectively means that different scenarios can be tested. These behaviours can be derived from the transition system and random explorations can be performed.

The second way for integration is to focus on the SUT and on its desired behaviours. Current test suites can be used to guide the selection of behaviours. However, in general, actions in the model will map to function calls, or shell command invocations in the case of operating systems. Technically for this to work new scripts will have to be developed. However, in most cases these new scripts are actually nothing more than small fragments derived from existing scripts. The main advantage of this approach is that much finer interleaving of operations will be possible.

We now illustrate the usefulness of the fine interleaving. Consider a test script that formats an storage device. In the process it also creates a partition with a file system and mounts it. As all the actions are in the one script there is only one possible test to run. However we can break up the operations in the script into the following parts:

- format a device

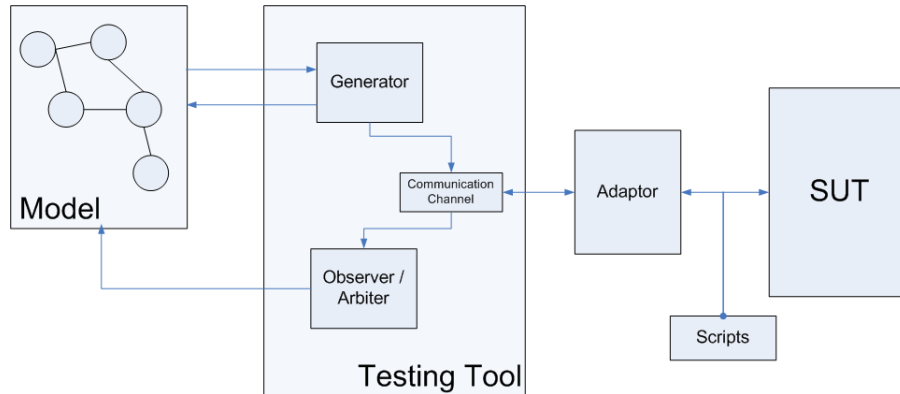


Figure 1. General architecture for a model-based testing approach

- partition the device with a variable size
- convert a partition into a file system and
- mount a file system.

Now, we can select different operations to execute the tests. For example, we can format a device and create two partitions on it. We can get one kind of file system into one partition, and a different one into another. We can later try to create a third partition that exceeds the size of the entire device, and see if this case is handled. Thus a number of behaviours that were not tested by the one script can be now be tested.

4. Case study

As an example of how model-based approaches can be used in the testing of operating system, and how this approach reuses current tests, we have implemented the testing of the storage HBA driver in the OpenSolaris operating system. We describe briefly the structure of the current testing framework used by the OpenSolaris community. Then, we describe the modelling of the storage management (sub)system for testing using an automated model-based testing tool, SmartMBT a testing tool developed by KJRoss and Associates². Finally, we also describe how the architecture of the current tests can be modified to capitalise advantages provided by the model-based approach.

²<http://www.kjross.com.au>

```
all
^fdisk
^format_all
^newfs

fdisk
/tests/diskcmd/fdisk/tc.fdisk

format_all
/tests/diskcmd/format/tc.format

format_part
/tests/diskcmd/format/tc.format{6,8,9}

newfs
/tests/diskcmd/newfs/tc.newfs
```

Figure 2. Test scenario specification

Current testing framework

The current tests for OpenSolaris storage management are contained into the Storage Driver Test Suite (SDTS)³ and use the Test Environment Toolkit (TET) as the underlying testing framework. These tests are structured in the following manner. Each *test suite* is made up of one or more *test cases*. Each test case is an executable program constructed by grouping together test functions, called *test purposes*. In general, a test case defines the order of execution of its test purposes. The test cases in a test suite are processed by the Test Case Controller (TCC) and the list of test cases that are to be processed by the *TCC* is specified in a *test scenario* file.

The code in Figure 2 was extracted from a test scenario file. This code specifies different scenarios that can be executed. For example, if the tests

³<http://www.opensolaris.org/os/community/storage/tests/>

```

:
if ! label_smi >> $logfile 2>&1; then
cti_report "FAIL: label smi on $stds_disk"
cti_reportfile "$logfile"
cti_fail "tp_newfs_001: FAIL"
return
else
cti_report "PASS: label smi on $stds_disk"
fi
:
if ! build_ufs >> $logfile 2>&1; then
cti_report "FAIL: create ufs on $rdev and mount it to $mp"
cti_reportfile "$logfile"
cti_fail "tp_newfs_001: FAIL"
return
else
cti_report "PASS: create ufs on $rdev and mount it to $mp"
fi
:

```

Figure 3. Test purpose code

are invoked for execution without specifying a particular scenario, the scenario *all* will take place and all test cases will be executed. However, if the scenario *format_part* is invoked then only test purposes 6, 8 and 9 of the test case *tc_format* will be executed.

When writing test cases in the TET framework, the tester is required to supply the test purpose code that actually executes the test operation. As an example, Figure 3 partially shows the code of a test purpose included in the SDTS. In this code, the auxiliary function *label_smi* is invoked which means that a previously identified disk (defined in variable *\$stds_disk*) will be formatted and a SMI label will be build into it. Later, the function *build_ufs* will format the first partition of this disk, convert it into a new file system and mount this file system into a predefined directory *\$mp*.

From an analysis of the code in Figure 3 we can conclude that the test purposes on this suite are written at a high level of abstraction. This is, the scripts do not correspond to the invocation of simple shell commands but to a structured sequence of invocations. We can also conclude that each test purpose has been designed to be executed independently. So each test purpose modifies the initial state of the system to enable its execution and they must “clean” the state of the system so that the next test purpose can be executed without problems. As a result, although the test case defines the order of execution, there

is really no pre-defined logical sequence between test purposes.

We now show how the behaviour of the scripts (as well as the desired behaviour of the system) can be captured as models for testing.

5. Model-based framework

Before we define how the current testing framework is integrated into a model-based framework, we describe some of the features of the tool (SmartMBT). Models in this tool are labelled transition system described in Prolog. To simplify the development of the models, all models have the following general structure.

The first part is the declaration of *state variables*. State variables are used to represent the states of the system along with their properties. A particular *state* in the transition system is defined as a mapping from the set of variables to the set of their possible values. In Figure 4 we show a partial view of the Prolog code that defines the state variables for our example of the HBA storage driver. The state variables for the example of the HBA storage driver include, a set of storage devices that can be connected to the system, a set of partitions for each device, and a set of mounting points to access the partitions. Some properties of these elements can also be modelled by boolean variables. These boolean variables are particularly useful when a unique device is tested (which is the case of the current framework). They serve to indicate whether a device is available or not, or if a partition is mounted, or, in the case of ZFS filesystems, if a pool has been already created. In essence, we have *partitions*, *diskAvailable*, *mountDirs* as the main state variables. They are lists (initialised to the empty list) as at any give time there may be many partitions, disks and mount points. The boolean values are used to indicate their current usage.

The second part is the definition of actions (or operations). Each action defines a parameter generation section and a transition definition section. The transition definition checks the current state of the system by getting the values of state variables. Then it checks that preconditions that en-

```

state_var(partitions, table([atom,int,atom,int])).
state_init(partitions) :- setv(partitions,[]).

state_var(diskAvailable, table([atom,bool,int,atom])).
state_init(diskAvailable):- setv(diskAvailable,[]).

state_var(mountDirs, table([atom,bool,atom])).
state_init(mountDirs) :- setv(mountDirs,[]).

state_var(zPoolCreated,bool).
state_init(zPoolCreated).
state_var(zfsMounted,bool).
state_init(zfsMounted).
:
:

```

Figure 4. State variables definition

```

params(fdisk3(Disk)):-
getv(diskAvailable,DiskAvailable),
member([Disk,true,_,_],DiskAvailable),
setv(diskAvailable,DiskAvailable).

transition(fdisk3(Disk)):-
getv(diskAvailable,DiskAvailable),

delete(DiskAvailable,[Disk,_,_],_DiskAvailableTemp1),
append(DiskAvailableTemp1,[[Disk,true,200,'smi']],_DiskAvailable),

_SmiLabel = true,

setv(diskAvailable,_DiskAvailable),
setv(smiLabel,_SmiLabel).

```

Figure 5. Model operations corresponding to the current STDS suite

able the operation are met. Finally, it declares changes in the values of state variables and stores them defining the resulting state after the operation. A state transition expressed as a Prolog rule is shown in Figure 5. The action `fdisk3` checks if a disk is available to be formatted, and formats it with a `smi` label.

In general the action labels in the model form a one to one map with the test purposes in the TET framework. For example, in Figure 5 the `fdisk3` action label corresponds to the test purpose 3 of the `fdisk` test case. In this case, the fact that each test purpose can be executed independently is central to achieve the “action - script” mapping.

After the state variables and the actions of the model have been defined, we use the SmartMBT tool to perform a random exploration over the model with the aim not only of producing test sequences but also of generating a graph that reveals the structure of the model. A snapshot of the SmartMBT tool interface is shown in Fig-

ure 6 present some of the features available to the user. A random exploration can be directed by the tester by using an interactive approach (`try` option). The random approach can also be fully automated and still be directed by assigning different “weights” to different actions. The “weight” of an action is a relative probability of execution, the greatest the weight greatest the probability. Actions can be avoided by assigning weight 0 to them. A pictorial representation of the STDS suite model, generated by random exploration, is shown in Figure 7.

Figure 7 shows that there are 10 states in our model (generated after executing 500 tests) with many transitions not really changing the state of the model. This is consistent with the behaviour of the current scripts where most of them “clean” the system before finishing, which means that the state of the system remains unchanged. However, the graph also shows that some scripts did not clean the resulting state, thus producing a new state in the model. Although generating the graphical view is not possible for large systems, the feature is useful when exploring a subsystem of the model.

Refined model structure

Our aim in refining the model is to provide more flexibility to the model-based approach in generating new testing scenarios by random and interactive exploration. Refining the model means then that we need more fine-grained actions or labels (and their corresponding scripts). Consider the test purpose in Figure 3, it can be broken up into actions `fdiskSMI`, `addPartition`, `createUFS` and `mountUFS`. Additionally, if we intend to really provide flexibility, actions such as `deletePartition`, `umountUFS`, and other actions that represent failure of the previous actions, such as `fdiskSMI_fail`, `mountUFS_fail` and so, need also to be included into the model.

In general an operation should fail if the state of the system does not allow it such as when some of the preconditions are not met. In the case of the formatting operation of a file system, we can format a device only if its file systems are not mounted. A (normal) test usually verifies that

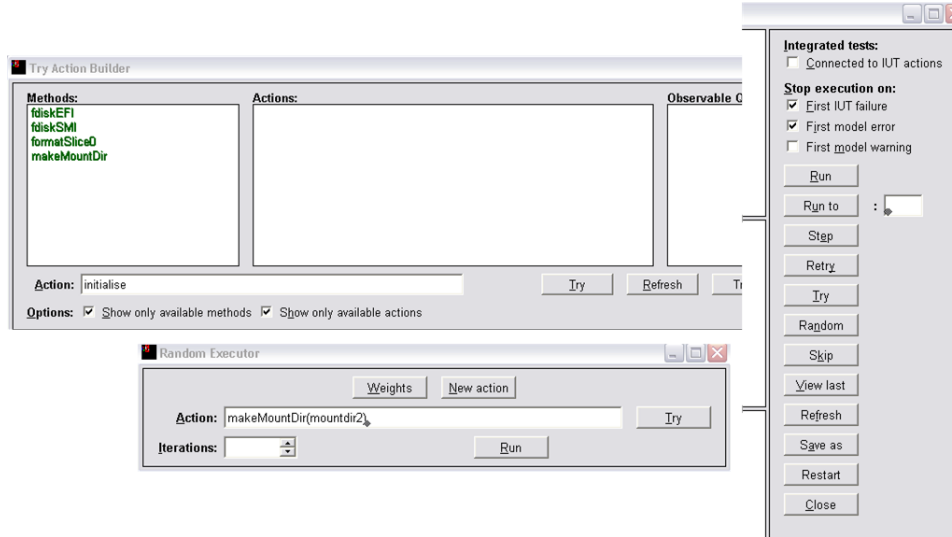


Figure 6. Graphical interface of SmartMBT tool

condition before executing the formatting action. Our failing test will, on the contrary, try to force a scenario where the formatting is executed over mounted partitions, therefore, expecting the action to fail.

Figure 8 shows a pictorial description of how the refinement of the models is done. Part (a) shows a single script (shown in Figure 3) that changes the state of the system from state q to q' . Part (b) shows the same test purpose decomposed into the operations cited in the previous paragraph. Not much changes except that now the test is a sequence of operations rather than a single one. Part (c) shows how the refined test purpose is capable not only of testing the same transition from q to q' as the original test purpose but also several other combinations like adding additional partitions, deleting partitions, mounting and unmounting partitions, formatting the disk in the middle of the transition to start all over again and even to fail on trying to format the disk because of a mounted partition. This clearly shows how different behaviours are tested.

Given this new fine-grained model, the result of a random exploration performed over the model shows an increase in the number of states. Our initial tests focused only in one of the five test cases included in the SDTS suite, the creation of a new

file system. Even in this reduced scope the number of states increased three times. The new model generated 30 states after performing the random exploration. In general it is difficult to predict the size of the complete model as it depends on the granularity of the new actions.

To change the model it is not enough to perform more flexible tests. The new model requires different scripts to be used by the Adaptor that connects to the SUT in order to execute the tests automatically. In general, these new scripts are not so different to the ones in the current suite but a break up of them. To illustrate this better, in Table 1 we show how the code of the script in Figure 3 is divided among the new actions of the model.

In those new scripts, some of the code from the original script has been removed. This is the case of the calls to the CTI framework, *cti_report* and *cti_fail*. The purpose of this code in the original script is to trap errors and provide a test report in the CTI/TET framework. We replaced these functions for a return code into the new scripts. Although this replacement is not necessary, it is convenient because it enables the Adaptor to determine when a script has failed and report it to the Arbiter in the testing tool indicating that the test had *failed*. Trapping errors as in the original

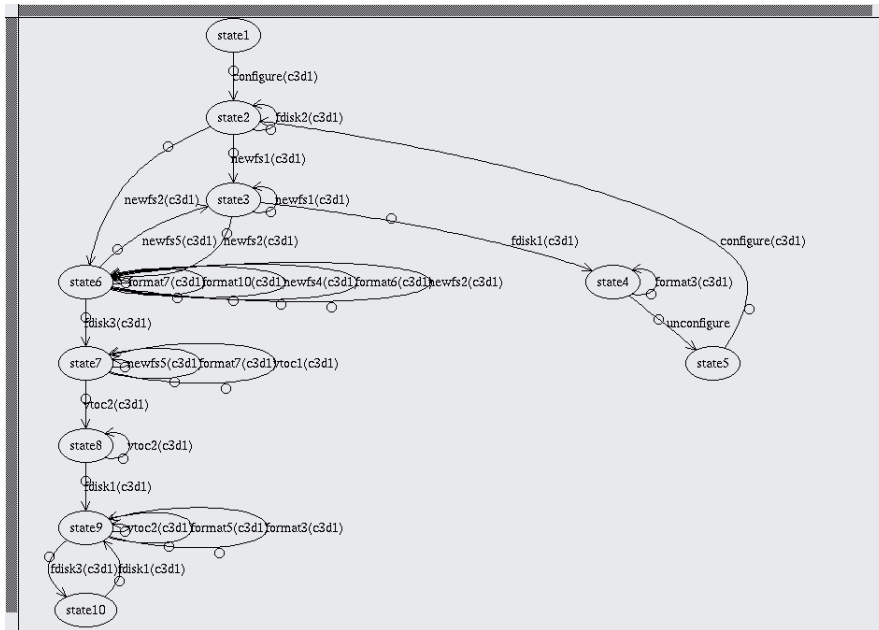


Figure 7. Model representation of current testing framework

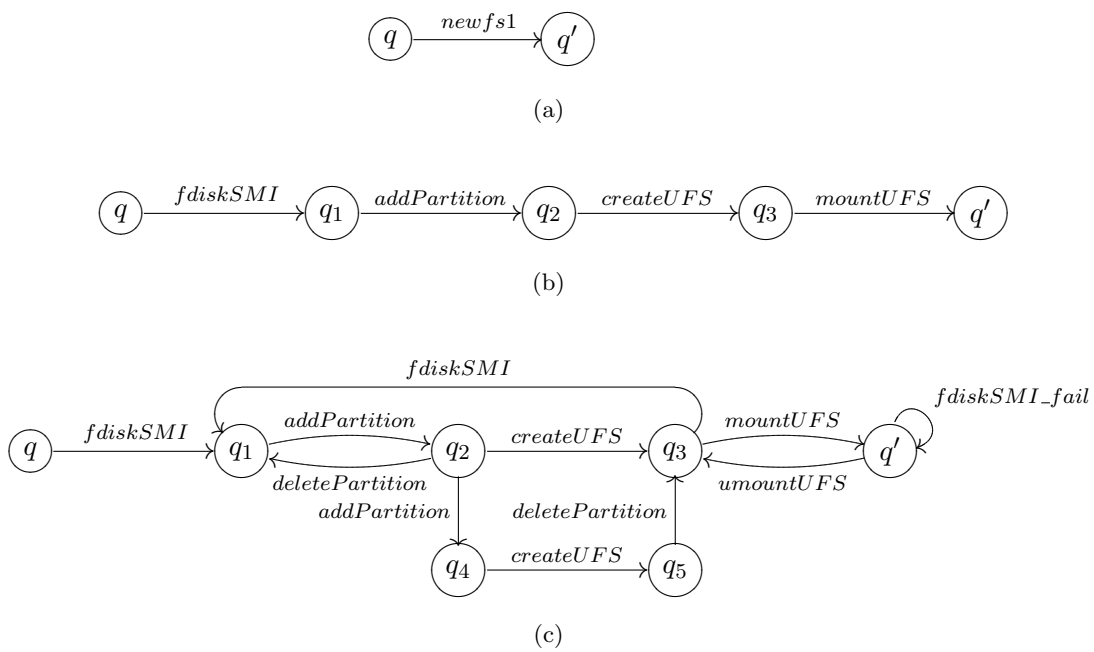


Figure 8. Refined models structure

Table 1. New model actions and their corresponding script code

fdiskSMI	<pre> if ! fdisk -B /dev/rdisk/\\${stds_disk}; then return 1 fi </pre>
addPartition	<pre> print "partition\n0\n\n0\n\${size}gb \ \n1\n\n0\n0\n2 \ \n\n0\n0\n3\n\n0\n0\n4\n\n0\n0\n5\n\n0\n0\n6\n \ \n\n0\n0\n7\n\n0\n0\nlabel\nq\nq" \ > \${testdir}/create_slice0.txt if ! format -f \${testdir}/create_slice0.txt \ -s \$stds_disk; then return 1 fi </pre>
createUFS	<pre> if ! yes newfs \$rdev >/dev/null 2>&1; then return 1 fi </pre>
mountUFS	<pre> if ! mount \$bdev \$mp >/dev/null 2>&1; then return 1 fi </pre>
umountUFS	<pre> if ! umount \$mp > /dev/null 2>&1; then if ! umount -f \$mp > /dev/null 2>&1; then return 1 fi fi </pre>

scripts will mean for the Adaptor that the scripts always terminate correctly and thus will indicate that the test *passed* although it produced an error report.

6. Discussion and conclusions

A model-based approach for testing operating systems and especially their kernels can provide an adequate way of documenting tests as well as a dynamic way of executing them. One of the advantages of writing models is that it can help to discover bugs in the code without even executing the tests. In the case of the current SDTS suite for the HBA storage driver of the OpenSolaris system we discovered a bug in the code of the test suite while writing the models that map to the current scripts. There was a problem with an auxiliary function that verifies the size of the storage device. There is a precondition for some operations that the device should be less than one terabyte. However this operation was failing when we tested a device of 500Mb. It was a case of an incorrect

negation and the test passed after we fixed the code.

Scripts in the current test suite pass the tests when the invocation to shell commands returns normally. This is logical but means that without further oracles to verify the resulting state of the system, the tests rely on the SUT itself to tell when an error or bug has been found. That clearly reduces the chances of catching “rare” bugs. The fact that scripts are quite independent, this is, they initialise their state and “clean” the resulting state means sequencing of scripts cannot be used to define oracles as it is usual in black box testing.

A model-based approach for testing operating systems can be implemented at different levels of abstraction. There exists a trade-off between abstraction and flexibility, the more abstract the model, less flexible to generate test scenarios. Consider, for example, the Sun’s ZFS file system. It is composed of two elements, *pools* and the actual *file systems*, which can be manipulated independently using different system commands. Nevertheless, we can decide that we want to test only

the operations related to the file systems without dealing directly with the pools. In this case, to create a ZFS file system would be a simple atomic operation that hides the need of creating the underlying pool and mounting the file system into an specified directory. However, testing situations such as if we can create two or more file systems into the same pool, or to check if we can create a pool without creating any file system on it, were not possible.

There also exists a trade-off between abstraction and model complexity. A low level model can be more flexible but the number of possible states can grow very large leading to the state explosion problem. In the example of the ZFS file system, if we model pools and their ability to host multiple file systems, the number of states will grow exponentially with respect to the number of file systems in the pool.

The definition of the model's level of abstraction is a design decision. As such it requires an understanding of both, the theory behind the models and the domain of the system that is being modelled.

Model-based approaches do not eliminate the need of testing code (test harnesses) to execute the tests into the real implementation. However, the exercise of writing the models can lead to discover bugs in the testing code. These approaches facilitate also the reuse of models and code. Thus we believe that the use of a model-based approach brings any test framework to the objective of using standard test suites to test different operating systems with certification purposes.

Live demo

A live demo using SmartMBT in an OpenSolaris environment will illustrate more details of this work as well as the workings of the model-based testing technique. At the conference we plan to demonstrate how random and interactive approaches can be used.

Acknowledgements

The authors thank KJRoss and Associates for providing free access to the SmartMBT tool as well as providing a special scholarship for the first

author. The authors also thank Cristina Cifuentes and Brenda Bai from Sun Microsystems for their encouragement.

References

- [1] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 73–88, New York, NY, USA, 2001. ACM.
- [2] S. Kandl, R. Kirner, and P. Puschner. Automated formal verification and testing of C programs for embedded systems. In *10th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, Santorin, Greece, May 2007.
- [3] G. Klein. Operating system verification — an overview. Technical Report NRL-955, NICTA, Sydney, Australia, June 2008.
- [4] K. Sen. Concolic testing. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 571–572, New York, NY, USA, 2007. ACM.
- [5] A. S. Tanenbaum, J. N. Herder, and H. Bos. Can we make operating systems reliable and secure? *Computer*, 39(5):44–51, 2006.
- [6] J. Tretmans. *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, chapter Model Based Testing with Labelled Transition Systems, pages 1–38. Springer Berlin Heidelberg, 2008.
- [7] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.