SOFTWARE TESTING, VERIFICATION AND RELIABILITY
*Softw. Test. Verif. Reliab.* 2012; **22**:33–63
Published online 5 March 2010 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/stvr.431

# A case study in model-based testing of specifications and implementations

Tim Miller[1,*,†] and Paul Strooper[2]

[1]*Department of Computer Science and Software Engineering, The University of Melbourne, Parkville, Vic. 3010, Australia*
[2]*The University of Queensland, School of ITEE, Brisbane, Qld. 4072, Australia*

SUMMARY

Despite the existence of a number of animation tools for a variety of languages, methods for employing these tools for specification testing have not been adequately explored. Similarly, despite the close correspondence between specification testing and implementation testing, the two processes are often treated independently, and relatively little investigation has been performed to explore their relationship. This paper presents the results of applying a framework and method for the systematic testing of specifications and their implementations. This framework exploits the close correspondence between specification testing and implementation testing. The framework is evaluated on a sizable case study of the *Global System for Mobile Communications 11.11 Standard*, which has been developed towards use in a commercial application. The evaluation demonstrates that the framework is of similar cost-effectiveness to the BZ-Testing-Tools framework and more cost-effective than manual testing. A mutation analysis detected more than 95% of non-equivalent specification and implementation mutants. Copyright © 2010 John Wiley & Sons, Ltd.

Received 2 June 2008; Revised 6 December 2009; Accepted 5 January 2010

KEY WORDS: specification animation; model-based testing; testgraphs

## 1. INTRODUCTION

Formal specifications can precisely and unambiguously define the required behaviour of a software system or component. However, formal specifications are complex artefacts that need to be verified to ensure that they are consistent and complete. Following IEEE terminology [1], the term *specification verification* is used to refer to any process that is used to determine whether the specification conforms to the requirements of the specification activity.

Several techniques for specification verification exist, including informal activities such as review and testing, as well as more formal techniques, such as theorem proving and model checking. In this paper, specification testing forms the basis of verification, which involves interpreting or executing a specification with the intent of finding faults. Like implementation testing, specification testing can only show the presence of faults in a specification and not their absence. However, as a more lightweight approach to specification verification and validation compared to model checking and theorem proving, it can be effective at revealing faults. The task of specification testing involves planning, selecting test inputs and expected outputs, executing tests, and comparing the actual output of the execution to the expected output. This is similar to the task of implementation testing, so it is reasonable to assume that some of the effort in specification testing can be reused during implementation testing.

*Correspondence to: Tim Miller, Department of Computer Science and Software Engineering, The University of Melbourne, Parkville, Vic. 3010, Australia.
†E-mail: tmiller@unimelb.edu.au

In the previous work [2, 3], the authors present a framework and method[‡] for the systematic testing of formal, model-based specifications using specification animation. At the core of this framework are *testgraphs*: directed graphs that partially model the states and transitions of the specification being tested. Testgraphs are used to model specifications, and then derive sequences for testing by repeatedly traversing the testgraph from the start node. Several generic properties that should hold for model-based specifications are identified, and a partially automated method that allows developers to check these properties during testgraph traversal is presented.

In later work [4], the authors discuss how to support the testing of implementations of specifications by using the animator to check that the outputs produced by an implementation conform to the specification. They also investigate how testgraphs can be used to drive the testing of graphical user interfaces (GUIs) [5]. The approach is applicable to any model-based specification language that has an appropriate animation tool, however, the tool support for the framework is implemented for the Sum specification language [6] and the Possum animator [7]. The case study presented in this paper also uses these languages and the corresponding tool support.

The main contribution of this paper is the application of the entire framework to the case study of the *Global System for Mobile communications* (GSM 11.11) [8], which is a file system with security permissions on a *Subscriber Identity Module* (SIM) for mobile devices. The goals of this case study are two-fold: to evaluate the cost-effectiveness of the framework on a sizable system that has been developed towards use in commercial software; and to demonstrate the fault-detection ability of the framework by having faults carefully seeded in the specification and two implementations by third parties and a mutant generator, and running the tests to ensure that the faults are detected. The evaluation demonstrates that the framework is of similar cost-effectiveness to the BZ-Testing-Tools [9] framework and more cost-effective than manual testing. A detailed mutation analysis demonstrates that the framework is effective at uncovering faults in the specification and both implementations. The mutation analysis detected more than 95% of non-equivalent specification and implementation mutants.

The paper also makes two other minor contributions. The first is the presentation and application of the complete framework for testing specifications and their implementations, which exploits the close correspondence between specification testing and implementation testing. Many of the ideas used in this framework are taken from standard software testing to make the framework intuitive for anybody with testing experience. In previous work, the authors have presented a framework and method for specification testing [3], a method for mapping a testgraph to an implementation and using a specification animator as a test oracle [4], and for mapping a testgraph to a GUI for the purpose of implementation testing [5]. This paper collates much of this work to present the entire framework for specification testing and implementation testing. The second minor contribution is the illustration of how the metric proposed by Legeard *et al.* [10] can be used to measure the coverage that a testgraph achieves of a specification, which is important for both specification testing and implementation testing.

This paper is organized as follows. The relevant background material is presented in Section 2, followed by a brief overview of the framework in Section 3. In Section 4, the GSM 11.11 requirements, specification, and implementations are presented. Sections 5 and 6 discuss the application of the framework to verify the GSM 11.11 specification and implementations, respectively. Section 7 presents an evaluation of cost-effectiveness and fault-detection ability of the framework, using the GSM case study. Related work is presented in Section 8, and Section 9 concludes the paper.

## 2. BACKGROUND

This section presents the Sum specification language [6] and Possum animation tool [7]. Sum is used to specify the GSM example, and Possum is used as the basis for the tool support in the framework.

[‡]From here, the term ‘framework’ will be used to mean ‘framework and method’.



DOI: 10.1002/stvr

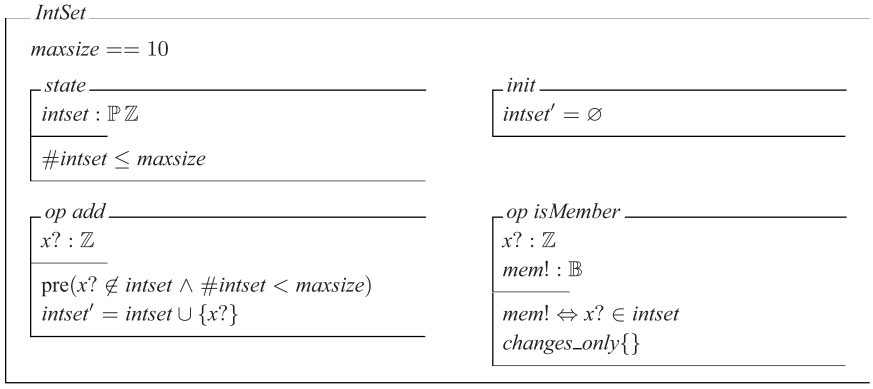



Figure 1. Sum Specification of *IntSet*.

### 2.1. Sum

Sum is a modular extension to the well-known Z specification language [11]. The most notable difference is that Sum supports modules, which provide a way to specify self-contained state machines within a specification. Each Sum module contains a state schema, an initialization schema, and zero or more operation schemas. Figure 1 shows an example Sum specification of a bounded integer set.

Sum uses explicit preconditions in operation schemas using the *pre* keyword. Like Z, input and output variables are decorated using ? and !, respectively, and post-state variables are primed ($'$). All operation schemas automatically include the primed and unprimed versions of the state schema in their declarations. A predicate of the form *changes_only A* in an operation postcondition states that the operation can only change the state variables in the set $A$. Therefore, the statement *changes_only*{} specifies that no state variables are allowed to change. No *changes_only* statement in an operation implies that all state variables can change.

The example in Figure 1 contains a state with only one variable, *intset*, which is a set of integers. The initialization schema asserts that the set is empty. There are two operations in this module: *add*, which adds an integer to the set if that integer is not already in the set and the set is not full; and *isMember*, which returns a boolean indicating whether an integer is in the set.

### 2.2. Possum

Possum is an animator for Z and Z-like specification languages, including Sum. Possum interprets queries made in Sum and responds with simplifications of those queries. A specification can be tested by executing specification operations in sequence, and Possum will update the state after each operation. For example, the following query adds the element 1 to the *intset* state variable in the *IntSet* module:

$$add\{1/x?\}$$

The syntax above represents binding; that is, the pair in the set $\{1/x?\}$ defines the binding of the variable $x?$ with the value 1. In this case, the value 1 is substituted for every occurrence of the input variable $x?$ in *add*. Using queries such as this allows users to execute specifications by executing queries, such as the one above, in sequence. The behaviour of the specification can be observed and compared to the expected behaviour. For example, one can initialize the *IntSet* specification, execute $add\{1/x?\}$, and then execute $remove\{1/x?\}$, and observe whether the system has returned to the initial state.

However, Possum is much more powerful than this. For example, it can evaluate the set of all elements for which *isMember* returns true:

$$\{x? : \mathbb{Z} | isMember\{true/mem!\}\}$$



DOI: 10.1002/stvr

To simplify queries such as this, Possum uses a mix of generate-and-test and other constraint-solving techniques (see Hazel *et al.* [7] for more details). Due to the undecidability of Z and Sum, it is clear that Possum cannot execute/evaluate every predicate or expression given to it. The results in this paper are applicable only to specifications that are executable.

Possum also supports Tcl/Tk plug-in user interfaces for specifications, which allows users not familiar with Sum to interact with the specification through a user interface. Possum defines a simple default user-interface for every specification tested, which displays the current binding for each state variable in the specification being tested.

## 3. OVERVIEW OF THE FRAMEWORK

This section presents an overview of the framework for testing specifications and implementations. The aim of the framework is to perform two activities: verify a specification using specification testing, and verify an implementation of that specification using implementation testing. By exploiting the similarities between specification testing and implementation testing, the overall testing effort can be reduced. For example, the tests executed on the specification should be executed on the implementation. Therefore, the tester can extend the test suite used for testing the specification when testing the implementation.

The framework is based on *testgraphs*: directed graphs that partially model the states and transitions of the specification being tested. Formally, a testgraph is a quadruple $(N, s, \Sigma, \delta)$, in which:

- $N$ is a finite, non-empty set of nodes, representing states of the system under test;
- $s$ is the start node, corresponding to the initial state of the system;
- $\Sigma$ is a finite, non-empty set of arc labels, representing state transitions of the system under test; and
- $\delta : N \times \Sigma \nrightarrow N$ is a partial function that defines the node that is reached when an arc is traversed from a source node.

The sets $N$ and $\Sigma$ are chosen so that the states and transitions that are of interest to testing are included in the testgraph. A testgraph is like a finite state machine, except that the nodes may represent more than a single state (they are abstractions of sets of states). The testgraph partially models the system under test, focusing on the states and transitions that should be tested. This model is then used to generate test cases for specification testing and implementation testing.

The tester performs the following tasks for the specification testing stage:

1. Derive a testgraph that models the states and transitions of interest to testing from the informal requirements using standard testing heuristics.
2. Measure the coverage that the testgraph achieves on the specification. If the testgraph does not achieve adequate coverage, improve it until it does.
3. Derive test sequences from the testgraph by generating paths through the testgraph starting from the start node and use these sequences to drive the testing of the specification using the Possum animation tool.

Once specification testing is complete, the tester must perform the following additional tasks to test an implementation:

4. Update the testgraph to take into account any new states or transitions that may need testing. This step considers the fact that the testgraph is modelled at a point when the data types and algorithms of the implementation may not have been designed. If special tests are required, for example, if a set data type in the specification is implementation as a binary tree, then the tester can use information about binary trees to improve the testgraph. At this stage, the tester can also deal with deferred or given sets that may not be known at the specification stage, but will be defined in the implementation.
5. Define a mapping between the specification interface, which uses the abstract specification language, and the implementation interface, which uses the concrete programming language.



DOI: 10.1002/stvr

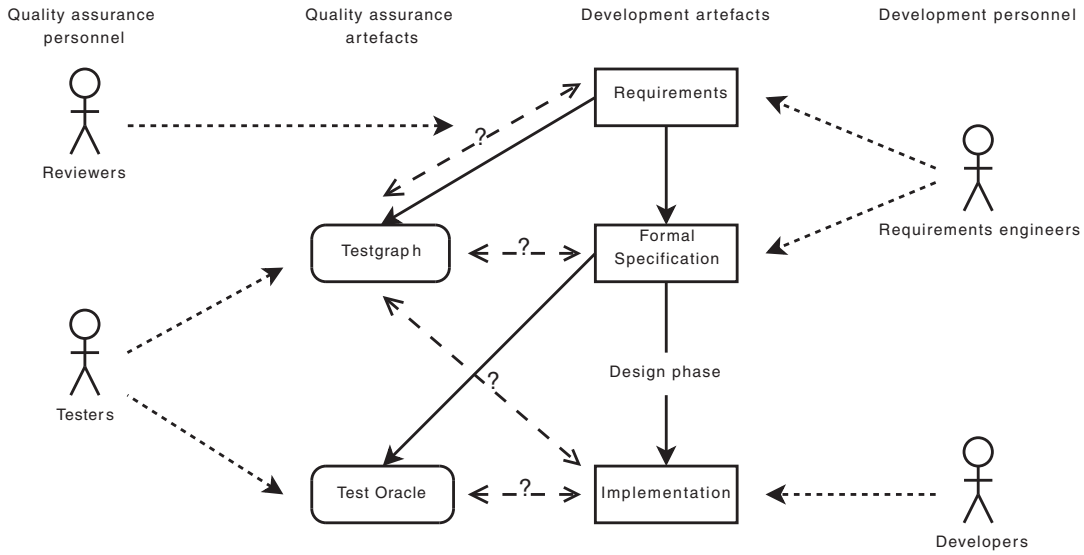



Figure 2. Overview of the framework identifying artefacts (boxes), verification activities (dashed arrows), roles (stick figures), and responsibilities (dotted arrows).

6. Similarly, define a mapping between the testgraph, which uses the abstract specification language, and the implementation interface, which uses the concrete programming language.
7. As for the specification testing, drive the testing of the implementation using test sequences generated from the testgraph. During traversal, the animator is used to check whether the actual outputs produced by the implementation conform to the specification.

Figure 2 presents a graphical overview of the framework. In this figure, the stick figures represent the roles in the framework, square boxes represent design artefacts, and rounded boxes represent verification artefacts. There are three types of arrows: solid arrows, drawn between artefacts, represent that the destination artefacts is derived from the source artefact; dashed arrows with question marks represent the activities in the framework, specifically, that the two artefacts are compared for consistency; and dotted arrows represent that the role at the source of the arrow is responsible for a derivation/activity.

From this figure, one can see that requirements engineers derive the specification from the informal requirements. Separately, testers derive a testgraph from the informal requirements. Some knowledge of the formal specification is required to derive the testgraph, such as operation names and inputs/output variable types, however, this is minimal and not shown in the figure. The verification activity between the testgraph and the specification indicates that the testgraph is compared with the specification for consistency. The activity between the requirements and the testgraph, performed by reviewers, indicates that testgraphs should be reviewed for correctness and completeness[§].

For verifying the implementation, one can see that the test oracle is derived from the specification, and that the testgraph is now used to test the implementation, rather than the specification. The testgraph may be modified to cater for implementation features, as discussed above. The test oracle is not reviewed because it is automatically derived from the specification.

### *3.1. Specification testing*

In specification-based testing (of implementations), the specification defines the behaviour to which the implementation should conform. The process of testing is then verifying whether or not the

[§]Clearly, other reviews are within the scope of development, such as reviews of the requirements, specification, and implementation; however, these are out of scope of the testgraph framework.



DOI: 10.1002/stvr

specification and the implementation define the same behaviour. The specification is therefore a *model* of the implementation.

Empirical evidence suggests that using an alternative implementation as a model for testing is not sufficient, because programmers tend to make the same types of faults in the same places, and therefore, there is a high probability that both implementations will contain the same faults [12]. Using a specification as a model, on the other hand, is more suitable because the specification is at a different level of abstraction (preconditions and postconditions) to the underlying program (program statements and branches). The specification is also more straightforward to derive, and therefore less error-prone, because it describes only the behaviour, and not how to achieve that behaviour.

Using testgraphs for specification testing follows a similar premise. In this approach, the artefact being verified is the specification, not the implementation, and the model of the artefact is the testgraph, not the specification. A testgraph is less likely to contain faults than a specification because it is only a partial model of the specification. Furthermore, testgraphs are similar to scenarios or test cases: they describe concrete examples of behaviour, but do not need to model all cases of that behaviour.

To derive the testgraph, the tester uses the informal requirements and standard testing heuristics to select states of the system that are of interest to testing. These states become *nodes* in the testgraph. Then, the tester selects transitions between these states, which become the *arcs* of the testgraph. The transitions are specified formally using the operations from the specification. For the nodes, two types of properties are used to check the behaviour of the specification at each node: *generic properties*, which are properties that should hold for every specification; and *application-specific properties*, which are the properties that are specific to the system being tested. Deriving application-specific properties is similar to the standard test case selection problem, whereas generic properties can be automatically defined for any specification. In previous work [3], the authors present a method for checking several generic properties of model-based specifications. These checks are:

- *An initialization check*: Check that there exists an initial state in the specification that satisfies the invariant;
- *A satisfiability check*: Check that there exist inputs, outputs, pre- and post-state values for each operation that satisfy that operation;
- *A precondition check*: Check that for each input/pre-state pair that satisfies an operation's precondition, an output/post-state pair exists that satisfies the postcondition and state invariant; and
- *A state invariant check*: Check that for each post-state that an operation allows (not including the state invariant), the state invariant holds on this post-state.

Using the BZ-Testing-Tools test coverage metric [10], coverage that a testgraph achieves on a specification is measured[¶]. Each operation in a specification is converted into a *control-flow graph* (not to be confused with testgraphs). The tests executed on that operation are analysed to measure how many paths of the control-flow graph are executed.

A control-flow graph is constructed by transforming the postcondition of an operation into its *effect predicates*. An effect predicate is a disjunction in the postcondition, with each disjunction corresponding to one *effect* of the operation. The disjunctions are obtained by re-writing the operation, similar to re-writing a predicate into disjunctive normal form (DNF). Like DNF, negations are pushed inwards until they are only applied to atomic predicates. Unlike DNF, disjunctions are not propagated to the outermost level; instead the re-writing is applied to the arguments of the disjunction. For example, the predicate $(A \wedge \neg(\neg B \wedge C)) \vee D$ is re-written by pushing the $\neg$ inwards until only negations of atomic predicates remain. This results in the predicate $(A \wedge (B \vee \neg C)) \vee D$.

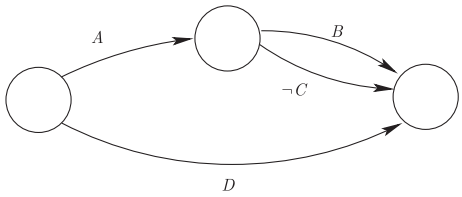
[¶]So far, coverage has been measured by hand, but Legeard *et al.* [10] discuss tool support for this that could be adapted to measure coverage automatically.



DOI: 10.1002/stvr

A control-flow graph is constructed from the effect predicate. Representing this information in the form of a graph improves readability and makes paths easier to identify when compared to the effect-predicate notation.

Constructing a control-flow graph from an effect predicate is done by creating a start node, and for each disjunction in the effect predicate, creating an arc to each of its arguments' graphs from the current node. Each conjunction is a new node connecting the two subgraphs of its arguments. For example, the representation of the predicate $(A \wedge (B \vee \neg C)) \vee D$ is the following:




Using the test sequences derived from the testgraph, the tester measures whether or not these sequences cause each disjunction corresponding to a branch to be executed. If this is not the case, the testgraph should be updated to achieve this coverage. Similar to code coverage, this criterion is viewed as a *necessary* condition, rather than a *sufficient* one. Using this as the sole measure of quality has the same problems as code-based coverage metrics in implementation testing, such as missing functionality not being detected. Considering this, it is advisable to measure the coverage achieved by the testgraph after it is completed, rather than using the coverage as a guide to producing the testgraph.

Once the tester is satisfied that the coverage is adequate, the specification is tested. Test sequences are generated using a depth-first search algorithm. A path is terminated when the search reaches a testgraph node in which all outgoing arcs have been visited previously. The search then backtracks to a node in which this property does not hold, after which it goes forward again to find additional test sequences. The depth-first search terminates when all arcs have been visited.

For this to be possible, the testgraph must be weakly connected (all nodes connected to all others, ignoring the direction of the transitions), and each node must be reachable from the start node.

The pseudocode for testing paths generated by the testgraph traversal is:

```
for each path P to be traversed
    initialize the specification using the animator
    for each arc A(S, D) in P, where S is the source and D the destination
        perform the transition for arc A using the animator
        check the properties for node D using the animator
```

To support the tester, a testgraph editor is provided, and traversal of the testgraph and execution of the tests is automated. Information regarding inconsistencies between the testgraph and the specification is collected, and scripts are provided for generating specification test code that automatically checks generic properties. Previous work discusses the specification testing framework in detail and present several case studies [3].

### *3.2. Implementation testing*

A problem that must be addressed in specification-based testing is the difference between the specification language and the implementation language. For example, the data types used in the specification language may not be supported in the implementation language. Typically, when the specification is used for test case generation, the abstract test cases then have to be refined into the implementation language. Conversely, any outputs produced by the implementation that must be checked by the specification must be mapped back to the specification language, which is typically accomplished through an abstraction function.

A tester using the framework in this paper is likely to encounter these problems because the framework reuses the specification testgraph to test the implementation. When testing the



DOI: 10.1002/stvr

specification, the testgraph is traversed and the operations are checked using the Possum animator. When testing the implementation, the testgraph traversal is driven by a traversal algorithm that is part of the framework and that calls a *driver* class that must be implemented by the tester. The driver class implements the transitions associated with the testgraph arcs and the checking of the application-specific properties associated with the nodes. Checking the generic properties generally only makes sense for the specification language because predicates such as precondition and state invariants, on which the generic properties are based, are typically not represented in implementations.

For implementation testing, the tester may also need to add new states or transitions to the testgraph that are specific to the implementation. For example, the data structures used in the implementation may introduce some interesting states that are not relevant or interesting for the testing of the specification, or are not covered by the testgraph.

Once the testgraph and driver are finalized, the testing can proceed by traversing the testgraph. Traversal is performed using the same algorithm as that used for specification testing. To check the behaviour of the implementation, the driver code checks the properties at the testgraph nodes by executing appropriate calls to the implementation under test, monitoring any outputs. These outputs are then translated into the specification language using abstraction functions, and are sent to the animator to be verified. Any inconsistencies between the specification and implementation are reported. Abstraction is used in this way to avoid the problems of non-determinism discussed by Hörcher [13].

In the previous work [4], the authors discuss a tool for automatic generation of 'wrapper' classes, which implements the interface of the implementation-under-test, and inserts calls to the animator before and after each operation that is executed to verify the behaviour.

### *3.3. Risks*

As with any software engineering process, there are risks in applying the testgraph framework. The risks in the development process, that is, going from requirements to specification to code, are out of the scope of this paper and are not discussed. The testgraph framework itself has several risks associated with it.

The first and most significant risk is associated with the derivation of the testgraph. It is possible that the testgraph could be incorrectly derived from the informal requirements. This risk is mitigated using a third-party review; that is, having the testgraph reviewed against the informal requirements by someone other than the requirements engineers or testers (the 'Reviewers' role in Figure 2). It is further mitigated by the fact that the testgraph is compared to the specification, so any faults in the testgraph will likely be revealed during specification testing. The residual risk is that both the specification and the testgraph coincidentally model the same behaviour incorrectly. This is further mitigated by having two different roles to derive the two artefacts. A similar risk exists between the testgraph and the implementation, and can be mitigated in the same way.

A second risk is that the testgraph is incomplete. The testgraph is only a partial model of the specification, so it may not model some important behaviour that should be tested. This risk is mitigated by measuring the coverage that the testgraph achieves of the specification. This process is mechanical, and can be performed by any role (ideally, it would be performed by a tool).

The third risk is related to the test oracle, and refers again to coincidental incorrectness. The test oracle is derived directly from the specification, and is used to verify the output from the implementation. If the specification and implementation both model the same behaviour incorrectly, the testing will not uncover this. This is not unlikely because the implementation and oracle are both derived from the specification. If the test oracle was derived from the informal requirements, this risk could be reduced because there would be a greater level of independence. However, deriving the test oracle automatically from the specification reduces the risk of human contamination significantly, so the authors believe that this is acceptable. This risk is further mitigated by first verifying the specification via testing (and review), before it is used to check the implementation.

The final risk relates to the executability of specifications. For a specification to be tested using our framework, it must first be executable. The Z specification language, on which Sum is based,



DOI: 10.1002/stvr

is not fully executable, so there is a risk that the framework may not be applicable to some parts of a specification. This risk can be mitigated by using only an executable subset of Z [14].

## 4. THE GSM 11.11 SYSTEM

This section outlines the requirements of the GSM system, and discusses the specification and implementations that were tested as part of the case study. The GSM was specified in B and presented by Bernard *et al.* [15]. A reduced B specification is available in that paper. A later version of this was hand-translated into Sum, and a few minor changes were made to ensure that the specification followed the GSM informal requirements[‖].

### *4.1. Goals*

The goals of this case study are two-fold. The first goal is to evaluate the cost-effectiveness of the framework on a sizable system that has been developed towards use in commercial software. This is evaluated by recording the amount of effort taken to apply the various tasks in the framework to the GSM 11.11 system, and comparing this to the design and development of a manually derived test suite and to the BZ-Testing-Tools framework. The data for these two other test suites is taken from Bernard *et al.* [15].

The second goal is to demonstrate the fault-finding effectiveness of the framework by having faults seeded in the specification and two implementations by third parties and an automated mutant generator, and running the tests to ensure that the faults are detected. Additionally, the tests are run over the initial versions of the specification and both implementations to determine how many naturally occurring faults are detected.

Many details of the case study are too large to include in this paper. The source for the GSM 11.11 case study can be downloaded from `http://www.csse.unimelb.edu.au/ tmill/tgf/` and the tests can be run. Due to copyright restrictions, the second implementation tested (the Javacard implementation discussed in Section 4.4) has not been made available.

### *4.2. Informal Requirements of the GSM*

The GSM 11.11 Standard is a smart card application for mobile phones which describes the interface between the *SIM* and the *Mobile Equipment* (ME). Only a subset of the GSM requirements were modelled by Bernard *et al.* [15]—the selection, reading and updating of files on the SIM, and the security access associated with them.

*4.2.1. Files* There are two types of files in the GSM standard:

- Elementary Files (EF), which contain the data. Data can be stored as a sequence of bytes (for *transparent* EFs), or a sequence of records (for *linear-fixed* EFs), with each record being a sequence of bytes.
- Dedicated Files (DF), which are directories and can contain EFs and other DFs. There is one special DF called the Master File ($mf$), which is the root directory.

In GSM 11.11, there are two DFs. One is the Master File, and the other is called $df\_gsm$, which is a child of the Master File. There are 6 EF files. Figure 3 shows the structure of the file system. DFs and EFs are prefixed with $df$ and $ef$, respectively.

The system maintains a current directory (a DF) and a current file (an EF). The user can change the values of these by selecting them explicitly. If a DF is selected, and that selection is valid, this becomes the current directory. No current file is marked as being selected after the

[‖]The B specification presented by Bernard *et al.* [15] abstracts away from some of the details in the GSM Technical Specification.



DOI: 10.1002/stvr

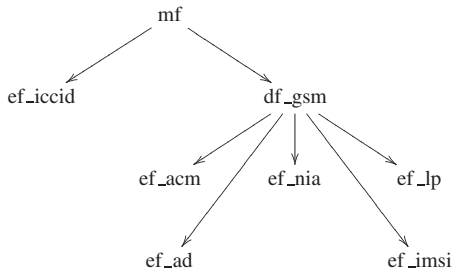



Figure 3. The directory structure of the GSM 11.11 SIM file system.

```
if ff = df_gsm or ff = mf
then
  if ff ∈ current_directory or
    current_directory ∈ ff or
    current_directory ∈ mf and ff ∈ mf or
    ff = mf
  then
    current_directory = ff and current_file = nothing
  else
    response is error code 9404
else
  if ff ∈ current_directory
  then
    current_file = ff
  else
    response is error code 9404
```

Figure 4. Pseudocode for *SELECT_FILE* Operation.

selection of a DF. If an EF is selected, and that selection is valid, this becomes the current file, and the current directory is unchanged. A selection is valid if one of the following conditions holds:

- the selected file is $mf$, the master file;
- the selected file is the current directory;
- the selected file is a DF and a sibling or parent of the current directory;
- the selected file is a child of the current directory.

The *SELECT_FILE* operation, used to select a new file or directory in the file system, is described by the algorithm outlined in Figure 4. *SELECT_FILE* is used throughout this paper to demonstrate the framework. In this figure, *ff* is the file that is selected, whereas *current_directory* and *current_file* are the current directory and file, respectively. Set notation is abused here**, with $ff \in mf$ representing that *ff* is a file in the directory called $mf$, and $ff \notin mf$ representing that *ff* is not a file in $mf$.

Once an EF is selected, the user can attempt to read or update that EF. The EF must first have the correct read or update permissions in order to be read or updated, respectively, (discussed in the next section). If the permissions are correct, the user can read or update a sequence of bytes to or from a transparent EF, or a specific record to or from a linear-fixed EF. If any problems occur during the selection, reading, or updating for a file, specific error codes are returned.

*4.2.2. Security* To read or update an EF, the file must have the correct access permissions. Each EF has its own access conditions for reading and updating. A *personal identification number* (PIN)

**This is not the way that the GSM is modelled, but this notation is used for simplicity.



DOI: 10.1002/stvr

can be presented to the SIM to grant new access permissions. The access rights corresponding to a PIN are blocked if three consecutive, incorrect PINs are presented to the SIM. This can be unblocked by presenting, to the SIM, a *personal unblocking key* (PUK). However, after 10 consecutive, incorrect PUKs, further access to certain files will be restricted.

There are four types of access conditions for EFs:

- Always (ALW): The action can be performed with no restrictions.
- Never (NEV): The action can never be performed through the SIM/ME interface. This action may be performed internally by the SIM.
- Administrator (ADM): Allocation of these levels and their requirements for fulfilling access conditions is the responsibility of the administrative authority.
- Card Holder Verification (CHV): The action is only possible if the correct PIN or PUK has been entered during the current session.

### *4.3. Specification of the GSM*

Several minor changes were made to the specification for this case study. Bernard *et al.* [15] generate test cases from specifications, and therefore only use data types needed for testing. To more accurately model the technical requirements of the GSM, some data types were expanded. For example, the possible values of the data stored on the SIM in the specification used by Bernard *et al.* are restricted to the enumerated set $\{d1, d2, d3, d4\}$, which are atomic constants representing blocks of arbitrary data. The tester in the BZ-Testing-Tools method must refine these into data before testing. In contrast, the Sum specification uses a sequence of bytes, in which $byte$ is defined as $-128 \ldots 127$. Even with these new data types, the translations of the B operations into Sum were straightforward.

The state of the GSM specification contains nine variables, outlined in Table I. The PUK cannot be changed, so it is not included in the state.

The formal model of the GSM standard contains nine operations, modelling the behaviour of selecting a new file or directory (*SELECT_FILE*), reading and updating files (*READ_BINARY*, *UPDATE_BINARY*, *READ_RECORD*, and *UPDATE_RECORD*), verifying and unblocking the CHV (*VERIFY_CHV* and *UNBLOCK_CHV*), returning the status of the SIM (*STATUS*), and resetting the card back to its initial state (*RESET*). Each of the operations except for *STATUS* and *RESET* return a four-digit number to indicate whether the operation was successful, and if not, why it was not successful.

Table II provides some metrics for the GSM specification (as well as the implementations, discussed in Section 4.4), including the number of constants, state variables, and operations defined in the specification, as well as the number of non-blank, non-comment lines. Table III presents the metrics for the operations in the GSM specification. The metrics 'State Changes', 'Atomic Evaluations', and 'Update Statements' refer, respectively, to the number of state variables that are changed by the operation, the number of atomic predicates that are evaluated by the specification

Table I. A description of the state variables in the GSM 11.11 specification.

| State variable | Description |
|---|---|
| *current_file* | Currently selected file |
| *current_dir* | Current directory |
| *counter_chv* | Number of remaining attempts to verify the PIN code |
| *counter_unblock_chv* | Number of remaining attempts to unblock the CHV |
| *blocked_chv_status* | Status of the CHV (blocked or unblocked) |
| *blocked_status* | Status of unblocking the CHV (blocked or unblocked) |
| *permissions* | Current access level to the 4 different types of file |
| *pin* | Current PIN code |
| *data* | Data contained in the files |



DOI: 10.1002/stvr

Table II. Metrics for the GSM 11.11 specification and implementation.

| Artifact | Constants | State variables | Functions/ operations | Lines |
|---|---|---|---|---|
| Specification | 23 | 9 | 9 | 324 |
| GSM-compliant implementation | 19 | 20 | 14 | 385 |
| Javacard implementation | 53 | 33 | 60 | 975 |

Table III. Metrics for the operations in the GSM 11.11 specification.

| Operation | State changes | Atomic evaluations | Update statements | Input variables | Output variables |
|---|---|---|---|---|---|
| *SELECT_FILE* | 2 | 8 | 6 | 1 | 1 |
| *READ_BINARY* | 0 | 3 | 8 | 0 | 2 |
| *UPDATE_BINARY* | 1 | 3 | 5 | 1 | 1 |
| *READ_RECORD* | 0 | 4 | 10 | 1 | 2 |
| *UPDATE_RECORD* | 1 | 4 | 6 | 2 | 1 |
| *VERIFY_CHV* | 4 | 3 | 10 | 1 | 1 |
| *UNBLOCK_CHV* | 5 | 3 | 10 | 2 | 1 |
| *STATUS* | 0 | 0 | 4 | 0 | 4 |
| *RESET* | 3 | 0 | 3 | 0 | 0 |

to define its control flow, and the number of statements that assign a value to an output or state variable. These metrics are intended to give some indication of the complexity of the GSM system.

### *4.4. The GSM implementations*

Unfortunately, the actual implementation of the GSM is not available. Instead, two different implementations were tested: one written by the authors from the formal specification, which complies to the standard and will be referred to as the *GSM-compliant implementation*; and one of a file system using EFs taken from the Sun's Javacard framework, which does not fully comply to the GSM standard and will be referred to as the *Javacard implementation*.

Table II provides some metrics to indicate the complexity of the two implementations. Note that parts of the Javacard implementation are related to functionality outside of the GSM 11.11 standard (approximately 20–25% of the code).

*4.4.1. GSM-compliant implementation* The GSM-compliant implementation is implemented in two Java classes: one for handling reading and writing of files, and one to handle file selection and security. Implementing this from the specification was straightforward and the structure of the implementation closely follows the specification. Efficiency, both in terms of time and memory, was not taken into consideration for the GSM-compliant implementation.

*4.4.2. Javacard implementation* Although the Javacard implementation does not implement the standard exactly, the authors believe that it enhances the case study because there is a greater level of separation between the Javacard implementation and the specification when compared to the GSM-compliant implementation, and because the Javacard implementation was not written by the authors of this paper.

The Javacard implementation came from a set of examples contained in Sun's Javacard framework[††]. An interesting aspect of this implementation is that differences between the implementation and the specification were identified, and then the test results were analysed to confirm that these were detected by the framework.

[††] See http://java.sun.com/products/javacard/.



DOI: 10.1002/stvr

The Javacard implementation is also written in Java. Because the implementation did not follow the standard, several changes were made to the implementation. The main changes were:

- Creating the files and their hierarchy. The original implementation had only two files, therefore, code was added to create the GSM files, specify their access levels, and the file hierarchy. This consisted of a single line of code per file.
- Checking the file hierarchy when selected. The original Javacard implementation had no file hierarchy, so code from the GSM-compliant implementation was used for checking whether a selection was valid. This consisted of approximately 10 lines of code.
- To use the same driver to test both implementations, an adaptor class with the same interface as the GSM-compliant implementation was placed around the Javacard implementation. This was the biggest change (200 lines of code), but most of this was trivially mapping method names and error codes in the implementation to those in the specification.

## 5. TESTING THE GSM SPECIFICATION

This section discusses the testing of the GSM specification using the framework. One difficult aspect of specification testing is the lack of a precise description of the intended behaviour. Any documentation of the intended behaviour tends to be written in prose. To help counter this problem, testgraphs are used to model a subset of the behaviour of a specification that is of interest to testing, so that the testgraph is less likely to have faults in it compared to the specification.

Recall from Section 3 that the tester performs three tasks for specification testing, which are discussed in Sections 5.1, 5.2, and 5.3, respectively.

### *5.1. Modelling the specification using a testgraph*

The state and operations of a specification provide important information about the selection of states to test. For example, the *READ_BINARY* operation will behave differently when there is no current file or when the current file is a binary file. Standard testing heuristics, such as partitioning and boundary-value analysis [16], are used to select special states for testing, and each state becomes a node in the testgraph. The heuristics used are independent of the framework.

For the GSM specification, which has an infinite state space, several such heuristics are applied to the state variables to produce a finite set of state values. For example, the number of unblock attempts is defined by a constant *MAX_UNBLOCK* in the specification, which is set to 10. Boundary-value analysis gives the values 0 and 10 for the boundaries, and 5 for a value between the boundaries. The *zero-one-many rule* gives the values 0, 1, and a value greater than 1, which 5 or 10 from the boundary-value analysis satisfy. Therefore, the testgraph includes states that represent 0, 1, 5, and 10 remaining unblock attempts.

The values chosen for each of the state variables are outlined as follows. For the current file, test no file, a current file for each of the respective file permission values, for both read and update, a transparent file, a linear-fixed file, a file in the *mf* directory, and a file from the *df_gsm* directory. These can all be covered using only four files. For the current directory, *mf* and *df_gsm* are the only possible values. For entering the PIN and PUKs, test 0, 1, and the maximum number of CHVs and unblock attempts. For the status of CHV, test the only two values: blocked and unblocked. For the access permission to files, the CHV files are the only ones that can change, so test both the true and false cases. For the PIN, test the initial value and one other value. For the content of the files, test the initial values, as well as a binary file that is empty, full, and in between, and a record file that is empty, full, and in between, as well as the first and last value of a record being updated.

Testing every possible permutation of these is impractical—it would result in over 50 000 different states, leading to a testgraph of unmanageable complexity. Testers can use standard partition combination techniques to combine multiple states into one. For the above example, note that the number of unblock attempts is unrelated to the current file, and vice versa, so a state from each partition is chosen and the number of states is reduced.



DOI: 10.1002/stvr

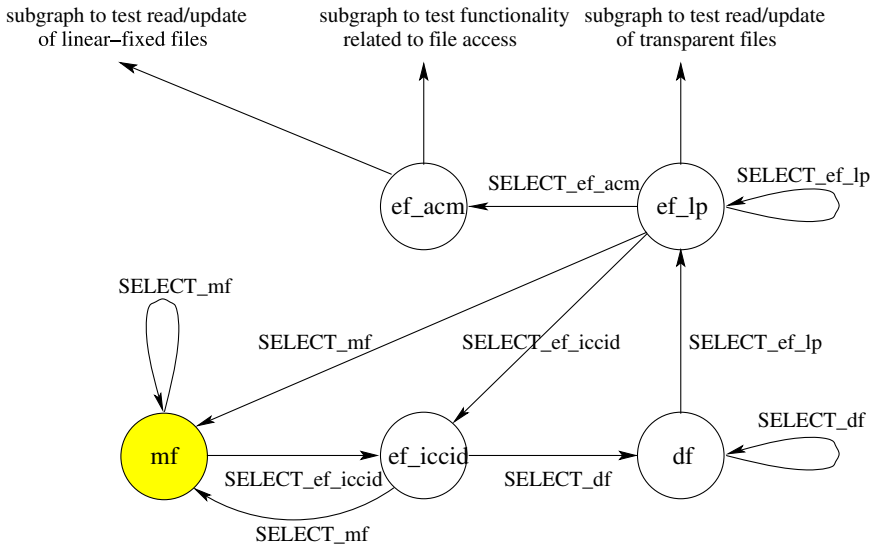



Figure 5. An excerpt from the GSM Testgraph for testing selection of file.

Once testgraph nodes are derived, arcs must be derived for the testgraph to be used as transitions during testing. The framework specifies that each node, except the start node, must have at least one arc leading into it, otherwise it will be unreachable. In practice, finding these transitions has not been difficult, although it requires a good understanding of the specification and the problem domain.

In addition, the labels of arcs on the testgraph, such as *SELECT_mf* in Figure 5, need to be related to the operations in the specification. Therefore, extra information is needed to model the behaviour of the specification precisely. The operations associated with the labels are defined as abbreviations. For example, the label *SELECT_mf* is defined as follows:

$$SELECT\_mf \mathrel{\widehat{=}} SELECT\_FILE\{mf/ff?, 9000/sw!\}$$

This means that the label *SELECT_mf* is a reference to *SELECT_FILE*, the value *mf* substituted for the variable *ff*? (the file to be selected) and the value 9000 substituted for the variable *sw*! (the return code indicating the selection is valid).

Specifying transitions and the resulting values at each node is not sufficient to test a specification, especially the operations in that specification that do not change the state and will never be used to specify a transition.

Additionally, a set of predicates is associated with each node to define the properties that should hold at that node—both the generic properties and the application-specification properties, as discussed in Section 3.1. Selecting the predicates to test the application-specific properties is similar to the test case selection problem for software testing. While specification-based testing approaches exist for selecting test cases from formal specifications for testing implementations of those specifications, this makes much less sense when those test cases are used to test the specification itself. Instead, typical heuristics for software testing are used, such as boundary-value analysis, by analysing the informal requirements of the system, as well as using tester intuition, and some information from the specification, such as the types of variables.

Table IV shows the predicates that are expected to hold at the $mf$ node in Figure 5. These predicates check whether the values of the state variables are correct, as well as the behaviour of operations, although never with input values that will cause a change in state—these are tested by the transitions. Excluding predicates testing the values of state variables at each node, each node contains an average of 6–7 predicates. Most of these were to check attempted reading or updating to or from a file, selecting a file, or checking the status.



DOI: 10.1002/stvr

Table IV. Predicates for the $mf$ node in the GSM testgraph.

| Predicate | Description |
|---|---|
| $current\_file = \{\}$ | There is no current file |
| $current\_directory = mf$ | The current directory is the $mf$ |
| *Similar checks for other state variables* | |
| $READ\_BINARY\{\langle\rangle / dd!, 9400/sw!\}$ | No file is selected, so reading a binary should return the error code 9400, and an empty sequence of data |
| $UPDATE\_BINARY\{\langle\rangle / dd?, 9400/sw!\}$ | No file is selected, so an update should return the error code 9400 |
| *Similar checks for READ_RECORD and UPDATE_RECORD* | |
| $SELECT\_FILE\{ef\_lp/ff?, 9404/sw!\}$ | The file $ef\_lp$ is not in the selected directory, so the error code 9404 should be returned |
| $STATUS\{mf/cd!, \ldots\}$ | The outputs of *STATUS* should be $mf$ for $cd!$ etc. |

Overall, the GSM testgraph contains 27 nodes, with 36 transitions between the nodes. It would be possible to reduce the number of nodes and arcs even further if several nodes were combined into one. However, the approach taken by the authors is to construct testgraphs from subgraphs that each test specific state variables or operations. The GSM testgraph separates the testing of the current directory and current file from the testing of reading and writing to files, which is again separate from unblock attempts.

The full GSM testgraph is too large to present in this paper, so instead, only an excerpt is shown in Figure 5. This subgraph is used to model the selection of different files in the GSM system; that is, to test the *SELECT_FILE* operation. Node labels represent the current file or directory, and arc labels document the file that is being chosen using the *SELECT_FILE* operation. The initial node is labelled *mf* and identified by the fact that it is shaded.

### 5.2. Measuring the coverage of a testgraph

Recall from Section 3 that coverage is measured using the BZ-Testing-Tool's test generation metric [10] based on control-flow graphs and effect predicates.

The control-flow graph for the *SELECT_FILE* operation from Figure 4 is shown in Figure 6. In this figure, the variables *current_file* and *current_directory* have been abbreviated to $cf$ and $cd$, respectively. This control-flow graph contains 14 paths. Out of these, only 6 are feasible. In fact, the arc labelled $ff \notin cd$ from node 2 does not contribute to any feasible paths because there are no values satisfying the subsequent arcs. This is due to the structure of the GSM file system. However, the testgraph covered only 5 of the 6 feasible paths. It was missing a case for when the file that was selected was the MF, and the current directory a DF. To add a test case to cover this path, one extra transition was added to the testgraph between two already existing nodes.

Metrics for the individual control-flow graphs of each of the operations are presented in Table V. Each of the remaining feasible paths are covered by the testgraph.

### 5.3. Using a testgraph to execute tests

The execution of tests on the specification is performed by traversing the testgraph repeatedly from the start node using a depth-first search until each arc is traversed. At each node during traversal, the properties associated with that node, such as those outlined in Table IV, are evaluated by the animator. If any of these evaluate to false, an inconsistency between the specification and the testgraph model has been identified.

Observability of relevant specification information has not been an issue in the case studies performed to date. Although state variables are encapsulated within modules, Possum allows state



DOI: 10.1002/stvr

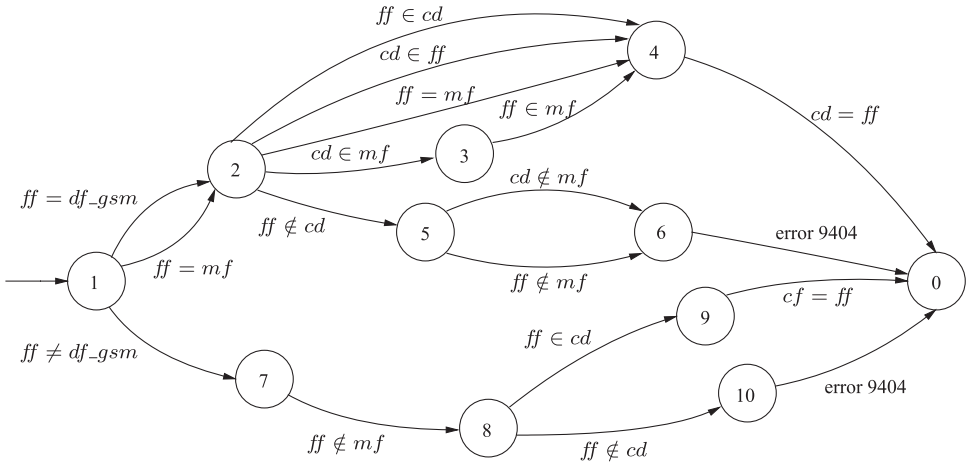



Figure 6. Control-flow graph for an abstract version of the *SELECT_FILE* operation.

Table V. Metrics for the size of the control-flow graph for the individual operations in the GSM specification.

| Operation | Nodes | Arcs | Paths | Feasible paths |
|---|---|---|---|---|
| *SELECT_FILE* | 11 | 18 | 14 | 6 |
| *READ_BINARY* | 8 | 9 | 4 | 4 |
| *UPDATE_BINARY* | 8 | 9 | 4 | 4 |
| *READ_RECORD* | 10 | 13 | 5 | 5 |
| *UPDATE_RECORD* | 10 | 13 | 5 | 5 |
| *VERIFY_CHV* | 8 | 10 | 4 | 4 |
| *UNBLOCK_CHV* | 8 | 10 | 4 | 4 |
| *STATUS* | 2 | 1 | 1 | 1 |
| *RESET* | 2 | 1 | 1 | 1 |

*CHECK_mf*

*state*

$report! : \mathbb{P}\{CURRENT\_FILE\_ERR, ..., STATUS\_ERR\}$

---

$current\_file = \{\} \Leftrightarrow CURRENT\_FILE\_ERR \notin report!$

$current\_directory = mf \Leftrightarrow CURRENT\_DIRECTORY\_ERR \notin report!$

...

$READ\_BINARY\{\langle\rangle/dd!, 9400/sw!\} \Leftrightarrow READ\_BINARY\_ERR \notin report!$

$UPDATE\_BINARY\{\langle\rangle/dd?, 9400/sw!\} \Leftrightarrow UPDATE\_BINARY\_ERR \notin report!$

$SELECT\_FILE\{ef\_lp, 9404/sw!\} \Leftrightarrow SELECT\_FILE\_ERR \notin report!$

$STATUS\{mf/cd!, \{\}/cf!, MAX\_CHV/cc!, MAX\_UNBLOCK/cuc!\} \Leftrightarrow$
$STATUS\_ERR \notin report!$

Figure 7. Operation *CHECK_mf* for the GSM.

variables to be read from outside the declaring module. For other animators and languages, the operation interfaces may be the only way to observe state variables.

Rather than manually sending each property to the animator, the framework relies on operations that automatically check the predicates for the current state of the specification. For example, the *CHECK_mf* operation, shown in Figure 7, is used to check the predicates for the *mf* nodes. This corresponds to the predicates defined in Table IV. Less interaction is required from the tester when using these operations. The variable *report*! in Figure 7 contains the set of all failures encountered, such as *CURRENT_FILE_ERR*, which are defined by the tester.



DOI: 10.1002/stvr

## 6. TESTING THE GSM IMPLEMENTATIONS

This section discusses the testing of the GSM implementations. Recall from Section 4 that two different implementations of the GSM are tested: the GSM-compliant implementation and the Javacard implementation.

The framework uses *self-checking implementations* for testing. These implementations are wrappers around the system under test that use Possum to verify that the behaviour of the implementation is consistent with that of the specification.

Recall from Section 3 that to test an implementation, the tester performs four tasks. In the GSM case study, there was no need to extend the testgraph for implementation testing, so the first step was skipped. The application of the remaining three steps to the GSM case study is described in Sections 6.1, 6.2, and 6.3 respectively.

### *6.1. Self-checking implementations*

In this section, the mapping between the specification and implementation for the GSM case study is described. This mapping is defined using self-checking implementations.

*6.1.1. Using an animator as a test oracle* Possum can be used to verify that a post-state and the output(s) are correct with respect to a pre-state and the input(s) for an operation. For example, one can send the following query to Possum:

$$SELECT\_FILE\{\{mf\}/current\_file, df\_gsm/ff?, \ldots, \{df\_gsm\}/current\_file', 9000/sw!\}$$

in which the ellipsis ('...') would be replaced by the remaining state variables. Each pair of the form $e/v$ represents the binding of the expression $e$ to the variable $v$. So this query checks whether, if the current file is *mf* ($\{mf\}/current\_file$) and *df_gsm* ($df\_gsm/ff?$) is selected, will the newly selected file become *df_gsm* ($\{df\_gsm\}/current\_file'$) and will the output code be 9000 ($9000/sw!$).

If this query fails—that is, the animator is unable to find a solution for the specified variable bindings—then it confirms that the post-state and output(s) are not correct with respect to the pre-state and input(s). If the post-state and output values were the result of executing the pre-state and input(s) on the implementation, then a failure to find a solution to the query demonstrates that there is an inconsistency between the implementation and its specification.

To make practical use of this, the sending of the query and the interpretation of the response are automated. This automation is achieved using a *self-checking implementation*. A self-checking implementation is a 'wrapper' class (an adaptor) around the implementation with the same interface as the class under test that executes operations in the implementation and checks the resulting behaviour using Possum. The wrapper returns any outputs from the implementation, thus displaying the same behaviour as the implementation itself. This is useful because it means that the self-checking implementation can be used in any testing framework, and at any level of testing; for example, the self-checking implementation for a single class can be used in place of that class during system testing.

For wrapper classes to communicate with Possum, a Java API has been defined to allow queries to be sent to Possum. The Java operation `isSatisfiable` checks whether an operation is satisfied with its variables bound to specific values. The use of this can be seen in Figure 8.

*6.1.2. Defining the wrapper class* The state of the wrapper class contains an instance of the class that communicates with Possum. The rest of the state is obtained by inheriting the class under test. This inheritance provides the wrapper class with access to protected variables declared in the class under test, which in some cases, makes reading the state variables more straightforward. More importantly, the inheritance allows the wrapper class to be used in place of the class under test in any place through the system.

To automate the checking of operations during testing, a precise mapping must be defined between an implementation and its specification. This includes defining mappings from the state of the specification to the state of the implementation, from the names of operations in the specification



DOI: 10.1002/stvr

```
public short SELECT_FILE(GSMFile ff)
{
  //abstract the pre-state
  ZPreState [] preState = abs();

  //make the call to super.SELECT_FILE
  short sw = super.SELECT_FILE(ff);

  //abstract the post-state
  ZPostState [] postState = abs();

  //abstract the value of the inputs and outputs
  GSMFile abs_ff = abs(ff);
  Short abs_sw = abs(sw);

  //collect the variables
  ZVar [] vars = {preState, ...,  abs_ff, abs_sw};

  //check using the animator
  TestResult result = animator.isSatisfiable("SELECT_FILE", vars);
  testLogger.handleResult(result);

  //return the output
  return sw;
}
```

Figure 8. `GSMWrapper.SELECT_FILE()`.

to the names of the operations in the implementation, and from the inputs and outputs of operations in the specification to the parameters and return values of the operations in the implementation.

Mapping operation and parameter names are straightforward. However, the differences in data types between specification languages and programming languages mean that the mapping between the states of the specification and implementation, as well as parameters of operations, is non-trivial. Hörcher [13] advocates abstracting from the concrete types of the implementation, because this eliminates problems with non-determinism. This approach is adopted here for this reason, and also because the animator can perform the checking. Therefore, the tester must implement *abstraction functions* to formalize the mapping for each system that they test.

The abstraction functions are written in the implementation language. To support this, a Java library has been implemented to give many of the types supported by Z (and therefore Sum). Classes for the basic Z types set, sequence, relation, function, and cross product have been implemented. These types are collections of Java objects, and as a result can be nested. Any Z given sets must be given a corresponding implementation.

*6.1.3. Testing operations* For each operation, a check is performed to confirm that the pre-state and input(s) do not violate the precondition, the post-state and output(s) are related to the pre-state and input(s) as specified, and the state invariant is maintained after each operation is executed.

To do this, each public operation in the class under test is overridden by the wrapper. Before calling the inherited implementation of the operation from the class under test (the superclass), the wrapper obtains the abstract pre-state using the abstraction function. Any outputs from the inherited operation are recorded during the execution of the operation. The abstract post-state is then obtained using the abstraction function. All of the variables, including state, input, and output variables, are collected, along with their values, and the animator is used to check whether the values satisfy the corresponding operation in the specification. Figure 8 shows the `SELECT_FILE` operation in the `GSMWrapper` class.

In this figure, a call to the inherited `SELECT_FILE` operation is made using the `super` keyword, capturing the output into the variable `sw`, and the pre- and post-states are retrieved before and after the call respectively. The `ZVar` array is the collection of variables, and is created from the input, pre-state, post-state, and output. When the input and output objects are created, their types are converted from the concrete types `GSMFile` and `sw` to their abstract types `Short` and `String`, respectively, with the `String` object being the name of the file. The `isSatisfiable` operation



DOI: 10.1002/stvr

is then called, the result handled by a test logging utility, and finally, the value of the output is returned.

This provides an example of abstracting inputs and outputs. The input and output types of operations in the specification can differ from those of the implementation. The values of those inputs and outputs must be abstracted to be able to compare the two using an animator. Figure 8 demonstrates the use of these abstraction functions. In this figure, the input to the operation is of type `GSMFile`. However, the state variable *current_file* is a set containing the name of the currently selected file. Therefore, an abstraction function is used to get the abstract value.

In other work [4], the authors discuss a tool that automatically generates wrappers using information from the specification combined with information from the implementation. The abstraction functions are implemented by the tester because these cannot be generated automatically as they are application dependent. To use the tool, the implementation must follow certain rules, such as declaring parameters in the same order as the specification.

### 6.2. Producing a driver class

The driver class is used to define a mapping between the testgraph and the implementation. In the framework, a driver class has three methods: `reset`, `arc`, and `node`. The `reset` method is called at the start of each path, the `arc` method is called when a testgraph arc is traversed, and the `node` method is called at each node. For the `arc` and `node` methods, the arc and node labels are parameters. The tester must implement each of these three methods for the driver class, and by checking the label, each method can make appropriate calls to the wrapper class to execute the self-checking implementation. For example, the label *SELECT_mf* from Figure 5 is an arc label, so when this label is passed to the `arc` method, the driver calls the `SELECT_FILE` method to select the $MF$ file in the self-checking implementation.

The most time-consuming aspect of testing the GSM implementation was generating the driver. However, the testing of the specification already gives the tester a good indication on how to implement the transitions, and which checks to perform at each node. For example, at the testgraph node $mf$, representing no currently selected file and $mf$ as the current directory, the tester can refine the predicates from Table IV into equivalent tests on the implementation. Using this process, generating the driver took approximately one person day of effort. This is a part of the framework that could be automated further. Provided that the mapping between the specification and implementation interfaces follows a specific format, the checking predicates and arcs in the testgraph could be automatically translated into the implementation language.

### 6.3. Executing the testgraph

Testgraphs can be automatically traversed in the framework. This is less error-prone than traversing the testgraph manually or deriving an automated test script to traverse a testgraph, and is significantly more efficient. To traverse the testgraph automatically, the driver class is used.

The traversal process is similar to that for executing a testgraph for specification testing. Each time an arc is executed, the `arc` method in the driver class is passed the label for the arc. Following this, the node label for the destination node is passed to the `node` method. Any time a new path is commenced, the `reset` method is called. These driver methods call the self-checking implementation, so any deviation in behaviour from the specification will be logged. A testgraph traversal tool has been implemented to traverse the testgraph automatically.

## 7. EVALUATION AND RESULTS

In this section, the evaluation and results of the case study are discussed. First, the cost-effectiveness of the framework is presented, and compared to that of the BZ-Testing-Tools framework [15] and a manually generated test suite, which have both been applied to the GSM case study. Second, the effectiveness of the framework at finding faults is evaluated and the results discussed. The section concludes with a discussion about testgraph correctness.



DOI: 10.1002/stvr

### 7.1. Cost-effectiveness

The cost-effectiveness of the framework was evaluated by measuring how long the activities in the specification testing and implementation testing life cycle took to complete in person days. Table VI presents a breakdown of the estimated costs in person days for the main tasks that were performed.

During the generation of the testgraph and the supporting specifications, no major problems were encountered. The testgraph derived for the specification is large, but by breaking it into several smaller subgraphs, maintenance should be straightforward. Deriving this took approximately person days of effort. From [15], note that the GSM formal specification took approximately 12 days to develop, which means that the time taken to test the specification is significantly less than the time taken to develop it. A point to note is that, for the GSM case study, the testgraph was not updated for the implementation testing, which would not always be the case.

*7.1.1. Comparison with other testing frameworks* Bernard *et al.* [15] recorded similar data for their application of the BZ-Testing-Tools framework to the GSM 11.11 case study. Bernard *et al.*'s case study was a controlled experiment in which they compared their automated test generation framework to that of a manually derived test suite, produced by the developers of the GSM 11.11 system. The results for this are shown in Table VII (taken directly from [15]).

These results indicate a significant reduction in the amount of effort spent using the testgraph framework over both the manual and BZ-Testing-Tools approaches. However, there are several reasons why the results are biased in our favour.

First, the version of GSM 11.11 that was tested in [15] contained more files on the SIM, additional access conditions of type CHV, and two additional operations. Adding these into the case study would have increased the time of all four activities reported in Table VI.

Second, the authors had read the paper by Bernard *et al.*, which contained the B formal model of the GSM 11.11. Although the requirements analysis and testgraph derivation were both performed based on the informal requirements, reading the B formal model would have helped in understanding of the requirements.

Third, it is likely that the interface of the actual GSM 11.11 implementation is different to the interface of the implementations that was tested in this study. This could account for some increase

Table VI. Testing costs in person days for testgraph framework on the GSM 11.11 case study.

| Testgraph framework and method design | |
|---|---|
| Requirements analysis | 5 days |
| Testgraph design | 2 days |
| Test driver design and coding | 1.5 days |
| Test wrapper design and coding | 3 days |
| Total | 11.5 days |

Table VII. Comparison of testing costs in person days for manual testing and the BZ-Testing-Tools framework on the GSM 11.11 case study.

| | |
|---|---|
| *Manual Design* | |
| Test design | 6 days |
| Test writing and coding | 24 days |
| Total | 30 days |
| *BZ-Testing Tools design* | |
| Writing B specification | 12 days |
| Test generation and coding | 6 days |
| Total | 18 days |



DOI: 10.1002/stvr

in effort due to complexity, although the actual interface is not publicly available, so this cannot be checked.

From the data in Tables VI and VII, requirements analysis and testgraph design data are grouped together to provide an estimate of 7 days for design. This corresponds roughly with the results from the manual design. The BZ-Testing-Tools framework took 12 days for the equivalent activity. This is not surprising, because test case generation in the BZ-Testing-Tools framework is automated, so the upfront effort of deriving the specification significantly reduces the effort of test case generation. While the testgraph framework and BZ-Testing-Tools framework both require the tester to derive a model—the testgraph and B/Z model, respectively—it is unsurprising that the testgraph model takes less effort, because it is typically a simpler model.

The data for test driver and wrapper design/coding is grouped together to provide an estimate of 4.5 days for coding. This is significantly less than the 24 days of manual design. In addition to the reasons discussed above, this reduction is likely due to three aspects of the testgraph framework. First, the testgraph is a *formal* model, so refinement into code is more straightforward. The use of abstraction functions to link the testgraph to the implementation is made simple by the inclusion of a library of abstraction functions. Second, only check predicates and arc transitions associated with the testgraph need to be refined, and not every test case, as would have been done in the manual design. This is a benefit of model-based testing in general. Third, although this has not been verified, it is estimated that the testgraph framework generates less test code than the other two approaches. Whether this reduces the quality of the test suite has not been investigated, because the manually derived test suite is not available.

The test coding for the BZ-Testing-Tools framework took approximately 6 days, which is similar to the testgraph framework. The extra 1.5 days could be accounted for by the issues discussed earlier.

The results in this section are positive for the testgraph framework. While the raw data suggests that it requires less effort than the BZ-Testing-Tools framework, it is likely that the factors discussed above are biased towards the testgraph framework. Despite this bias, the results demonstrate that the testgraph framework is cost-effective when compared to other parts of the development process, such as specification, and when compared to other testing methods.

A common criticism of graph-based testing methods is that deriving and maintaining the graph is costly. Using a formal specification to derive the graph is one option, however, this requires a formal model to be written. The results in this section indicate that deriving a formal model is considerably more costly than deriving a testgraph.

A corollary of modelling the system using testgraphs, rather than formal specifications, is that the testgraph can be used to test the specification as well as the implementation. This is a key aspect of the testgraph framework. While the tasks in Table VII are tasks associated with implementation testing, Table VI includes the tasks necessary for specification testing as well. The results indicate that the effort required to test both the specification and the implementation is not much greater than testing the implementation alone. Testgraphs are not required to benefit from this—manually derived test cases can be applied on both as well. The BZ-Testing-Tools framework, or any specification-based testing method for that matter, cannot be used to test the specification, because test cases are derived directly from the specification itself, and will therefore always agree with the specification.

*7.1.2. Discussion* The results in this section demonstrate that the testgraph framework is cost-effective for the GSM case study, both in relation to the other parts of the development process, and when compared to two other testing approaches. Further controlled evaluation would be needed to identify the cost differences between the testgraph framework and the BZ-Testing-Tools framework.

### *7.2. Fault-finding effectiveness*

This section analyses the testgraph framework's effectiveness at detecting faults in programs. First, the faults that were uncovered in the specification and both implementations are discussed. Second, the results from a detailed mutation analysis are presented.



DOI: 10.1002/stvr

In addition, the Clover test coverage measurement tool[‡‡] was used to measure statement, method, and branch coverage.

### *7.2.1. Natural faults*

*7.2.1.1. Specification testing.* Two faults were uncovered in the specification, which were both made during the translation from B to Sum. The low number of faults is most likely because the specification was translated from a well-verified B specification.

One fault was the use of the read permission of files when attempting to update files, and the other was the incorrect classification of the read permission of a file, which was denying read access to a file that should have been readable. Both of these are classified as application-specific faults.

*7.2.1.2. Testing the GSM-Compliant Implementation.* While testing the GSM-compliant implementation, which had not previously been tested, a fault was discovered in the abstraction function of the wrapper class. In addition, 7 faults were found in the implementation. The faults included assigning the wrong access level to a file, null object exceptions, and forgetting to close output streams after updating a record, which meant that the record was not updated.

The wrong access level fault is a serious problem. The access to a particular file was always being granted for reading and was granted for updating after presenting the PIN, whereas the access should have never been granted for either read or update. This is a security breach that could have negative effects on the operation of the SIM.

The tests achieve 94.7% combined coverage of the GSM-compliant implementation. Of this, one branch is not executed because it is located in an unreachable part of the code. This corresponds to the infeasible path in the control-flow graph in Figure 6, as discussed in Section 5.2, so this cannot be covered by any test case.

The rest of the uncovered code is related to exception handling. These exception handlers catch input/output exceptions, which cannot be controlled via the interface. There is also no specification of the behaviour should an input/output exception occur, so tests for this must be derived separately.

*7.2.1.3. Testing the Javacard implementation.* As discussed earlier, there are some aspects of the GSM standard that the Javacard version did not implement (no GSM-specific EFs and no file hierarchy). Before testing, one other discrepancy was identified: when an `UNBLOCK_CHV` operation is performed, the CHV status is not reset to true, therefore files with CHV access cannot be read/updated in all cases for which they should be. Both of these were detected by the tests.

During testing, two additional discrepancies were discovered. The first is that when an `UNBLOCK_CHV` operation is performed, the CHV blocked status is not unblocked. This is similar to the CHV status not being reset to true, but in this case, the CHV can never be successfully verified. The second difference, which occurred in several places, is that the access privileges are being checked before the type of file. For example, if an attempt is made to add a record to a transparent file that does not have the correct access level, the response code will report that the file access is not satisfied, whereas the specification states that the response code should report that the file is of the wrong type. These faults are most likely related to the implementation being for a different standard, rather than mistakes made by the developer.

In addition to these two differences, a fault was uncovered in the implementation. When a record is updated and the string of bytes used to form the new value is longer than the existing value, there is not enough memory allocated to fit the new bytes. This fault does not seem related to the differing standards, but rather an oversight by the developer.

Assessing the coverage on the Javacard implementation is difficult. Owing to the fact that much of the code in the implementation is unrelated to the GSM specification, large parts of the code are untested, therefore, percentage statistics of the code covered are largely uninteresting. An analysis

[‡‡] See http://www.atlassian.com/software/clover/.



DOI: 10.1002/stvr

of the coverage results showed that the tests achieve 100% combined coverage of the code that is relevant to the GSM standard.

*7.2.2. Mutation analysis* In this section, the results of mutation analysis applied to the specification and implementation are presented.

*7.2.2.1. Manual Mutation Analysis of the Specification.* To evaluate the effectiveness of the approach with respect to fault finding, mutation analysis was used. A third party was asked to create 10 mutants of the GSM specification, each with one syntactic change to the original. These mutants were intended to represent realistic faults. Each of the new mutants were then run through the GSM specification testing process to see if the changes would produce runtime failures. This was done without inspecting where the changes had been made, so as to not influence the process.

Table AI in Appendix A outlines the changes that were made to the specification. Specification testing produced failures for each of the changes in the table, except for number 1. Mutant 1 was located in a part of the *SELECT_FILE* operation that is unreachable, corresponding to the infeasible path discussed in Section 5.2, and therefore, is equivalent to the original. All other mutants were killed using application-specific tests.

*7.2.2.2. Semi-Automated Mutation Analysis of the Specification.* Clearly, the above process has a human bias, in that the mutants were seeded manually. To counter this, a second mutation analysis was performed, in which mutants were automatically generated. As far as the authors are aware, no mutant generator exists for Z/Sum, so instead, the GSM specification was translated by hand into Java, using a Z toolkit written in Java. This translation was performed on a syntactic level; that is, the resulting implementation did not fulfil the behaviour in the specification, but is merely used to generate mutants. MuJava [17] was then applied to this implementation, and the mutants were mapped back onto the specification. The authors believe this to be a valid way to generate mutants for the specification, because mutants, by definition, are generated only at a syntactic level.

MuJava produced over 600 mutants of the GSM. Owing to limited tool support, this is too many to test (there exists no framework for executing collections of specifications automatically in Possum). Instead, a systematic selection of these was performed.

MuJava applies a series of *mutation operators* to a program. For example, one mutation operator takes a relational operator, such as <, and replaces it with all other relational operators, such as <=, ==, etc., resulting in several mutants. This operator is applied to every relational operator in the program, resulting in many different mutants.

For each of the nine mutation operators that are applicable to the GSM implementations, three mutants were selected. MuJava produces mutants in a sequence, so the first non-equivalent mutant was selected, the last non-equivalent mutant, and the non-equivalent mutant that was closest to the median of first and last. This resulted in a total of 23 mutants[§§]. The results of the mutation analysis are presented in Table VIII.

Table VIII shows that one mutant was not killed. Investigation demonstrated that mutants were created that changed the access privileges of individual files. The testgraph does not attempt to read and write to every file on the SIM. Instead, it only tests one file of each type of file permission (that is, one file each of type CHV, ADM, NEV, and ALW). The mutation that was not killed was the changing of an access privilege for a file that was not checked. The actual GSM system contains even more files than the ones in this case study, so testing each file would be even more complex, but clearly feasible, given that it is a finite number. This is a problem with the test selection procedure, rather than the framework itself. Clearly, more exhaustive testing may produce better results, but this is at a cost of complexity and effort.

[§§]Some mutant operators produced fewer than three non-equivalent mutants.



DOI: 10.1002/stvr

Table VIII. Results of mutation analysis on the GSM 11.11 specification for the testgraph framework.

| GSM specification | |
|---|---|
| Mutants killed | 22 |
| Non-equivalent living mutants | 1 |
| Total mutants | 23 |
| Mutation score | 95.65% |

*7.2.2.3. Manual Mutation Analysis of the Implementations.* As with specification testing, the implementation testing was evaluated using a combination of manually and automatically generated mutants.

Again, a third party was asked (different from the person who seeded faults in the specification) to create 10 mutants of both implementations, each with one small syntactic change.

Tables AII(a) and (b) in Appendix A outline the manually generated mutants of the GSM implementations. In Table AII(a), there are two implementations equivalent to the original. Mutant 1 is equivalent because, while the assignment is outside of the 'try/catch' statement, it was originally contained in the 'try' part, and the 'catch' part catches an exception that is not specified, so cannot be tested. Mutant 2 is equivalent because the behaviour is not changed at all by modifying the access level.

In Table AII(b), there are again two mutants equivalent to the original implementation, each an insertion of a unary operator in front of a constant definition. Because the name of the constant is used to reference its value in all parts of the code, and because the values were chosen only to be different from other constants (and remain so), these mutants are equivalent. All other mutants were detected.

*7.2.2.4. Automated Mutation Analysis of the Implementations.* For the automatically generated mutants, a full mutation analysis was performed using MuJava [17]. MuJava was used to generate all mutants of both implementations. To remove as much bias as possible from the evaluation, the mutant log created by MuJava was manually inspected before the tests were run to determine equivalent mutants, rather than analysing living mutants after the tests were run. Each mutant was then tested using the testgraph framework, and a mutation score was calculated. Any remaining mutants were assessed to see whether they were either equivalent mutants that had not been detected using manual inspection, or whether the test suite did not detect them. The results of this analysis are presented in Table IX.

The gross mutation score is the number of mutants killed, whereas the net mutation score is the number of *non-equivalent* mutants killed. The net mutation score is high, but not 100%. From Table IX, one can see that for the GSM-compliant and Javacard implementations, respectively, there were 8 and 25 non-equivalent mutants that were not killed. All of these mutants relate to only two omissions from the testgraph. First, as explained above, the testgraph does not attempt to read and write to every file on the SIM. This is the same problem with the testgraph that failed to kill the mutant in the specification testing.

Second, the testgraph tests only two different PINs and PUKs: a correct PIN/PUK, and an incorrect PIN/PUK. Several mutants were created that changed conditions such as `if (PUK == code_unblock)` to conditions `if (PUK >= code_unblock)`. The testgraph does not test a PUK that is greater than the code input by the user; only a correct and an incorrect (less than the input) PUK.

The are a large number of equivalent mutants for the Javacard implementation because a lot of the code is unreachable.

*7.2.3. Discussion* Overall, the results in this section indicate that the testgraph framework is effective at finding faults. All pre-identified faults were uncovered, as well as some additional faults. The mutation score is not 100%, however, the authors do not believe that the non-equivalent



DOI: 10.1002/stvr

Table IX. Results of mutation analysis on the two GSM 11.11 implementations for the testgraph framework.

| | |
|---|---|
| *GSM-Compliant Implementation* | |
| Mutants killed | 449 |
| Equivalent mutants | 76 |
| Non-equivalent living mutants | 8 |
| Total mutants | 533 |
| Gross mutation score | 84.24% |
| Net mutation score | 98.9% |
| *Javacard Implementation* | |
| Mutants killed | 619 |
| Equivalent mutants | 307 |
| Non-equivalent living mutants | 25 |
| Total Mutants | 951 |
| Gross mutation score | 65.1% |
| Net mutation score | 96.12% |

mutants that were not killed demonstrate any underlying issue with the testgraph framework. The testgraph could be extended to detect these mutants as well, but it is not clear how cost-effective these extensions would be.

### *7.3. Testgraph correctness*

It has been noted already in this section that approximately two person days of effort were required to derive the testgraph. However, the testgraph was not correct the first time. Despite uncovering two faults in the specification, the specification testing uncovered a handful of failures during the testing; that is, discrepancies between the testgraph and the specification.

In all but two of these cases, the testgraph was incorrect, rather than the specification. The reason for this is clear: the testgraph had not come under the level of scrutiny as the original B specification. This supports the assumption in Section 3.3 that the testgraph should be reviewed, like other work products in software development. In the case of the GSM case study, the testgraph was reviewed solely by its author.

It is important to note that it was more straightforward to locate a fault in the testgraph than to locate faults in the specification, which is unsurprising given the simple nature of testgraphs.

A further result regarding testgraphs is that no additional faults were detected in the testgraph during implementation testing. That is, all discrepancies between the testgraph (and specification) and the implementations were as a result of faults in the implementations. This result is attributed to the fact that the testgraph had come under significant scrutiny during the specification testing. This supports the hypothesis (see Section 3.3) that faults in the testgraph are likely to be uncovered during specification testing.

This result also indicates that no faults were uncovered in the test driver class. The authors attribute this to the fact that the driver class was derived from the information in the testgraph.

## 8. RELATED WORK

This section discusses related work on animation and testing, including work on using animation for the purpose of testing.

### *8.1. Animation*

There are several animation tools that automatically execute or interpret specifications such as PiZA [18], Possum [7], Pipedream [19], the B-Model animator [20], the ProB animator [21], the



DOI: 10.1002/stvr

VDMTools interpreter [22], and the CLPS-BZ solver [23]. Of these tools, the VDMTools interpreter is the only tool that supports coverage measurement, although details of how this coverage is measured are not provided by Fitzgerald *et al.* [22].

Kazmierczak *et al.* [19] outline one of the few approaches for systematic specification testing using Pipedream. The approach consists of three steps: performing an initialization check; verifying the preconditions of schemas; and checking a simple reachability property. The framework presented in this paper extends the approach proposed by Kazmierczak *et al.* by defining new properties to check, and by providing a specific method for checking these properties, including tool support.

Liu [24] also discusses how to use specification testing tools. Liu uses specification testing to test proof obligations. These proof obligations are tested to check that they hold for a subset of possible traces of the specification. Additional tool support for the method is not discussed. In later work [25], Liu presents a tool that extracts all functional scenarios from SOFL specifications using data-flow criteria. From these scenarios, sequence charts are derived and executed as test cases on the specification using the SOFL animator. Because the test cases are generated from the specification itself, this method is used for validating that the specification meets the user requirements, but would be less useful for testing correctness with respect to the specifier's understanding of the requirements.

Several different options were evaluated to measure coverage of testgraphs, including the control-flow and data-flow techniques proposed by Zweben and Heym [26], and the mutation analysis method proposed by Ammann and Black [27]. From this evaluation, it was concluded that the only specification-based technique that achieved good coverage of the specification was complete path coverage in control-flow analysis, but this is far too time-consuming even on small examples, and hence would not be practical for larger specifications.

### *8.2. Testing*

*8.2.1. Specification-based testing* Formal specifications can aid in testing implementations by providing a starting point for selecting test inputs, executing test cases and determining expected outputs. Research in this area includes work on generating test cases for individual operations [28–30], generating finite state machines for the purpose of testing [31–33], generating test oracles [13, 34, 35], and frameworks for specification-based testing that support test case generation, execution, and evaluation [36, 37].

The approach for using a formal specification as a test oracle outlined in this paper builds on the work of McDonald and Strooper [34] and Hörcher [13]. Both of these papers present methods for writing passive test oracles by using the information in a formal specification. These oracles test whether the precondition and postcondition of the specification hold on the inputs, outputs, pre- and post-state of the implementation. Abstraction functions are used to relate the data types of the implementation to those of the specification. Similarly, Aichernig [38] generates C++ code from VDM specifications for use as a test oracle. The approach in this paper for oracle generation differs from these because an animator is used as the underlying oracle. This is advantageous because the translation between languages is more straightforward, requiring only abstraction functions, and tools specifically designed for the specification language will be better equipped to check whether the behaviour is satisfied.

*8.2.2. Graphs and finite state machines* Testgraphs are similar to *finite state machines* (FSMs), except that the nodes in a testgraph can represent sets of states, rather than individual states.

A lot of work has been done on testing using FSMs. Dick and Faivre [31] generate FSMs from formal specifications. They generate FSMs by retrieving the pre- and post-states from test cases generated by partitioning schemas into *disjunctive normal form* (DNF), and using them as the states of FSMs. A transition is created between two states if the two states can be related via an operation. The FSM is then traversed, with every branch executed at least once. A number of other approaches exist that extend Dick and Faivre's work [32, 33, 39]



DOI: 10.1002/stvr

Leuschel and Turner [40] automatically generate graphs from B specifications, which are then reduced to a human-readable size. Grieskamp *et al.* [41] automatically generate graphs that are in turn used for testing C# implementations. A model of the implementation is written in Spec#, a superset of C# that contains higher-level constructs and data types, as well as pre- and post-conditions. A finite state machine is automatically generated from the model, and traversal algorithms are used to generate test suites from the state machine.

The GSM case study demonstrates that manually derived testgraphs can have errors. Automation could reduce the risk of this. The approaches discussed above for automatic generation of FSMs from specifications are useful for implementation testing, however, their applicability to specification testing is debatable. While it would be possible to generate FSMs, and therefore testgraphs, directly from the formal specification, this would be of little use for specification testing. The testgraph is a model of the expected behaviour of the specification, and any faults in the specification would be reflected in the testgraph as well. For example, if the GSM specification contained a fault that gave the user an unlimited number of attempts to unblock the SIM, then any FSM or testgraph generated automatically from the specification would also contain this fault. As a result, the fault would never be uncovered using specification testing. If the testgraph is created manually from the informal requirements, there is a risk of introducing a fault in the testgraph. However, the likelihood that the same fault occurs in the specification is low, and it is likely that the fault in the testgraph will be detected during testing.

Utting and Legeard [42] discuss ModelJUnit, a tool for generating test sequences from finite state machines in the JUnit test framework. The ModelJUnit notion of finite state machine is similar to that of testgraphs. ModelJUnit provides several algorithms for traversing the state machines, such as random traversal and arc coverage.

Hoffman and Strooper [43, 44] generate test cases for C++ classes by automatically traversing a testgraph using *Classbench*. These testgraphs are usually derived manually, without the aid of a formal specification. Later work by Carrington *et al.* [36] describes generating Classbench testgraphs from FSMs. States for these FSMs are derived by using the *Test Template Framework* [30] to specify sets of test cases, and extracting pre- and post-states from these test cases. Transitions are added between each node if possible, and the FSM is converted into a testgraph.

In this paper, testgraphs are used to drive both specification and implementation testing, but rather than derive testgraphs from FSMs, the tester defines the testgraph manually.

*8.2.3. Testing using animation* Legeard *et al.* [9] present an automated specification-based testing technique for testing Z and B specifications, called the *BZ-Testing-Tools* method. They use the specification to generate states to test, and then use a constraint solver to search for a sequence of operations that reaches each state. The behaviour of the implementation is compared against the simulation of the specification. As discussed in the previous section, this approach cannot be used to test specifications, because the expected behaviour is derived directly from the specification itself. The BZ-Testing-Tools method has been applied to the GSM case study [15], and their results are compared to the testgraph framework in Section 7.

Aichernig *et al.* [45] discuss validating the expected outputs of test cases by comparing them with the outputs from the IFAD VDM interpreter. They revealed problems in both the specification and implementation of an air-traffic control system using this approach. There does not appear to be automated support for this.

Khurshid and Marinov [46] use the Alloy Analyser to both aid with the generation of test cases, and to verify that an input/output pair is correct with respect to its correctness criteria — a relation between the input and output space. It is not clear how much support exists for automating this.

## 9. CONCLUSION

This paper presents the results of applying a framework for model-based testing of specifications and their implementations. Test execution is driven by the traversal of testgraphs: directed graphs that



DOI: 10.1002/stvr

model a subset of the states and transitions of the specification being tested. Testgraph traversal is automated, and test oracles for implementation testing are automated using a specification animator.

The testgraph framework is demonstrated and evaluated on a sizable case study of the *Global System for Mobile communications* (GSM 11.11) [8], which is the file system on a *SIM* for mobile devices. This case study was used to evaluate the cost-effectiveness of the framework and the fault-detection ability for specifications and implementations. For implementation testing, the results show that the framework was similar in terms of cost-effectiveness when compared to the BZ-Testing-Tools method [15], but much more cost-effective than manual testing (saving approximately 50% of the cost). The mutation analysis showed that more than 95% of non-equivalent specification and implementation mutants were detected by the framework in this case study.

The framework is built upon standard software testing ideas to make it appealing to practitioners. The authors believe that this framework increases not only the value in using specification testing, but in using formal specification as a basis for software development as a whole.

There are two notable directions for future work. The first is a more thorough investigation into modelling of testgraphs to consider the level of abstraction/independence between the testgraph and the specification. Empirical evidence suggests that testing using a model of a program that is at the same level of abstraction as that program is ineffective [12].

The second involves further automation of the framework. At present, some generic activities are automated, such as specification-level checks for generic properties and generation of the oracle. However, other parts of the process that are currently performed manually could be automated. The two most costly tasks identified for automation are measuring coverage of the specification and the production of the test driver class. The BZ-Testing-Tools framework uses the coverage metric to automatically generate test cases, so tool support for coverage measurement is clearly feasible. The driver class interprets the labels on the testgraph, and the checks associated with the testgraph are used to identify how to interpret these. With a systematic mapping between the specification and the implementation, these checks could be at least partially translated into code to provide much of the driver. Automating these two steps would reduce both the cost and risk of applying the testgraph framework.

## APPENDIX A: TABLES SUMMARIZING MUTANTS

The changes that were made to the manual mutation analysis of the specification are given in Table AI. The details of the manually generated mutants of the GSM implementations are given in Table AII.

Table AI. Mutants of the GSM specification.

| Mutant # | Description | Result |
|---|---|---|
| 1 | A directory can be selected if it is the sibling of the current directory or a child of the MF, rather than both | equiv. |
| 2 | Negate the evaluation of whether a specified EF is a valid selection | found |
| 3 | The contents of a non-deterministically chosen file are returned when *READ_BINARY* is invoked with no current file | found |
| 4 | Updating a binary file results in an additional value for the file, rather than overwriting the previous value | found |
| 5 | New records cannot be added to the end of a linear-fixed file | found |
| 6 | Administrative access to files is not permitted | found |
| 7 | The system is not blocked until 'maximum attempts +1' attempts have been made | found |
| 8 | The system is blocked after the first failed attempt to enter the PIN | found |
| 9 | Read and update permissions are swapped in *READ_BINARY* and *UPDATE_BINARY* | found |
| 10 | The initial directory is *df_gsm* instead of the master file, *mf* | found |



DOI: 10.1002/stvr

Table AII. Manually generated mutants of the GSM implementations.

| Mutant # | Description | Result |
|---|---|---|
| (*a*) *GSM-Compliant Implementation* | | |
| 1 | Move the assignment of a return code to outside of a `try/catch` statement | equiv |
| 2 | Modify access to a method from protected to private (required a change to the wrapper class to compile) | equiv |
| 3 | Negate a boolean in the conditional of a `for` loop, such that the data to a binary file is not written successfully | found |
| 4 | Access an 'out-of-bounds' record | found |
| 5 | Finish the search for a record index too early | found |
| 6 | Return the incorrect error code | found |
| 7 | Update data to the incorrect index of a record file | found |
| 8 | Increment the incorrect loop counter, such that records are not updated with all of the data | found |
| 9 | Return a non-existent error code | found |
| 10 | Remove braces ({ and }) of `else` branch such that an incorrect error code is returned | found |
| (*b*) *Javacard Implementation* | | |
| 1 | Increment an integer after it is used instead of before, causing memory for a record to be allocated after the record is written | found |
| 2 | Remove braces ({ and }) of an `else` branch | equiv |
| 3 | Access a record at the incorrect index | found |
| 4 | Replace a <= operator with a < operator such that an incorrect PIN is accepted | found |
| 5 | Force the precedence of an addition over a subtract such that the data is initialized incorrectly | found |
| 6 | Negate a boolean in a branch statement, therefore accepting an incorrect PIN and rejecting a correct PIN | found |
| 7 | Remove a `break` statement, so all PINs are treated as incorrect | found |
| 8 | Removal of a `break` statement, such that all files are classed as being of the incorrect category (e.g. EF instead of DF) | found |
| 9 | Allocate more space than necessary to store the PUK code | equiv |
| 10 | Remove braces ({ and }) of `if` branch such that a correct PIN is sometimes treated as incorrect | found |

ACKNOWLEDGEMENTS

The authors thank Tim McComb and Steve Phelps for taking the time to seed faults in the GSM specification and implementations, respectively. We also thank the anonymous referees for their comments on the earlier drafts of the paper.

REFERENCES

1. IEEE Computer Society. IEEE 1012-2004: IEEE standard for software verification and validation. 2004.
2. Miller T, Strooper P. Model-based animation using testgraphs. *International Conference on Formal Engineering Methods*. Springer: Berlin, 2002; 192–203.
3. Miller T, Strooper P. A framework for the systematic testing of model-based specifications. *ACM Transactions on Software Engineering and Methodology* 2003; **12**(4):409–439.
4. Miller T, Strooper P. Supporting the software testing process through specification animation. *First International Conference on Software Engineering and Formal Methods*. IEEE Computer Society: Silver Spring, MD, 2003; 14–23.
5. Miller T, Strooper P. A case study in specification and implementation testing. *Proceedings of the Asia–Pacific Software Engineering Conference* (*APSEC'04*). IEEE Computer Society: Silver Spring, MD, 2004; 130–139.
6. Kazmierczak E, Kearney P, Traynor O, Wang L. A modular extension to Z for specification, reasoning and refinement. *Technical Report 95–15*, Software Verification Research Centre, February 1995.
7. Hazel D, Strooper P, Traynor O. Possum: An animator for the SUM specification language. *Proceedings of Asia–Pacific Software Engineering Conference and International Computer Science Conference*. IEEE Computer Society: Silver Spring, MD, 1997; 42–51.
8. European Telecommunications Standards Institute. *GSM 11.11 v7.2.0 Technical Specification*, France, 1999.



DOI: 10.1002/stvr

9. Legeard B, Peureux F, Utting M. Automated boundary testing from Z and B. *Formal Methods Europe*. Springer: Berlin, 2002; 21–40.
10. Legeard B, Peureux F, Utting M. Controlling test case explosion in test generation from B and Z specifications. *The Journal of Testing*, *Verification and Reliability* 2004; **14**(2):81–103.
11. Spivey J. *The Z Notation*: *A Reference Manual* (2nd edn). Prentice-Hall: Englewood Cliffs, NJ, 1992.
12. Knight J, Leveson N. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering* 1986; **12**(1):96–109.
13. Hörcher HM. Improving software tests using Z specifications. *Proceedings of the Ninth Annual Z User Meeting*, Bowen JP, Hinchey MG (eds.). Springer: Berlin, 1995; 152–166.
14. Breuer PT, Bowen JP. Towards correct executable semantics for Z. *Z User Workshop*. Springer: Berlin, 1994; 185–209.
15. Bernard E, Legeard B, Luck X, Peureux F. Generation of test sequences from formal specifications: GSM 11.11 standard case-study. *The Journal of Software Practice and Experience* 2004; **34**(10):915–948.
16. Myers GJ. *The Art of Software Testing*. Wiley: New York, 1978.
17. Ma Y-S, Offutt J, Kwon Y-R. MuJava: An automated class mutation system. *Journal of Software Testing*, *Verification and Reliability* 2005; **15**(2):97–133.
18. Hewitt M, O'Halloran C, Sennett C. Experiences with PiZA, an animator for Z. *ZUM'97*: *The Z Formal Specification Notation*. Springer: Berlin, 1997; 37–51.
19. Kazmierczak E, Dart P, Sterling L, Winikoff M. Verifying requirements through mathematical modelling and animation. *International Journal of Software Engineering and Knowledge Engineering* 2000; **10**(2):251–273.
20. Waeselynck H, Behnia S. B-Model animation for external verification. *Proceedings of the International Conference for Formal Engineering Methods*. IEEE Computer Society: Silver Spring, MD, 1998; 36–45.
21. Bendisposto J, Leuschel M. A generic flash-based animation engine for ProB. *Proceedings of the B 2007*: *Formal Specification and Development in B*, *7th International Conference of B Users* (*Lecture Notes in Computer Science*, vol. 4355). Springer: Berlin, 2007; 266–269.
22. Fitzgerald J, Larsen P, Sahara S. VDMTools: Advances in support for formal modeling in VDM. *ACM SIGPLAN Notices* 2008; **43**(2):3–11.
23. Ambert F, Bouquet F, Chemin S, Guenaud S, Legeard B, Peureux F, Vacelet N. BZ-TT: A tool-set for test generation from Z and B using constraint logic programming. *Proceedings of the Formal Approaches to Testing of Software*, *Workshop of CONCUR'02*, Brno, Czech Republic, 2002; 105–120.
24. Liu S. Verifying consistency and validity of formal specifications by testing. *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems*. Springer: Berlin, 1999; 896–914.
25. Liu S, Wang H. An automated approach to specification animation for validation. *Journal of Systems and Software* 2007; **80**(8):1271–1285.
26. Zweben SH, Heym WD. Systematic testing of data abstractions based on software specifications. *The Journal of Software Testing*, *Verification and Reliability* 1992; **1**(4):39–55.
27. Ammann P, Black P. A specification-based coverage metric to evaluate test suites. *International Journal of Reliability*, *Quality and Safety Engineering* 1999; **8**(4):275–300.
28. Gaudel M. Testing can be formal too. *Proceedings of TAPSOFT'95*. Springer: Berlin, 1995; 82–96.
29. Stepney S. Testing as abstraction. *Z User Meeting'95*. Springer: Berlin, 1995; 137–151.
30. Stocks P, Carrington D. A framework for specification-based testing. *IEEE Transactions on Software Engineering* 1996; **22**(11):777–793.
31. Dick J, Faivre A. Automating the generation and sequencing of test cases from model-based specifications. *Formal Methods Europe*: *Industrial-Strength Formal Methods*. Springer: Berlin, 1993; 268–284.
32. Hierons RM. Testing from a Z specification. *Software Testing*, *Verification and Reliability* 1997; **7**(1):19–33.
33. Turner C, Robson D. A state-based approach to the testing of class-based programs. *Software—Concepts and Tools* 1995; **16**(3):106–112.
34. McDonald J, Strooper P. Translating Object-Z specifications to passive test oracles. *Second International Conference on Formal Engineering Methods*, Liu S, Staples J, Hinchey MG (eds.). IEEE Computer Society: Silver Spring, MD, 1998; 165–174.
35. Antoy S, Hamlet D. Automatically checking an implementation against its formal specification. *IEEE Transactions on Software Engineering* 2000; **26**(1):55–69.
36. Carrington D, MacColl I, McDonald J, Murray L, Strooper P. From Object-Z specifications to Classbench test suites. *Journal on Software Testing*, *Verification and Reliability* 2000; **10**(2):111–137.
37. Doong R, Frankl P. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology* 1994; **3**(2):101–130.
38. Aichernig B. Automated black-box testing with abstract VDM oracles. *Workshop Materials*: *VDM in Practice*! *Part of the FM'99 World Congress on Formal Methods*, Fitzgerald J, Gorm Larsen P (eds.). ACM Press: New York, 1999; 57–66.
39. Chang KH, Liao SS, Seidman SB, Chapman E. Testing object-oriented programs: From formal specification to test scenario generation. *The Journal of Systems and Software* 1998; **42**(2):141–151.
40. Leuschel M, Turner E. Visualising larger state spaces in ProB. *Proceedings of ZB 2005* (*Lecture Notes in Computer Science*, vol. 3455). Springer: Berlin, 2005; 6–23.
41. Grieskamp W, Tillmann N, Veanes M. Instrumenting scenarios in a model-driven development environment. *Journal of Information and Software Technology* 2004; **46**(15):1027–1036.




DOI: 10.1002/stvr

42. Utting M, Legeard B. *Practical Model-based Testing*: *A Tools Approach*. Morgan-Kaufmann: Los Altos, CA, 2007.
43. Hoffman DM, Strooper PA. The testgraphs methodology—automated testing of collection classes. *Journal of Object-Oriented Programming* 1995; **8**(7):35–41.
44. Hoffman DM, Strooper PA. ClassBench: A methodology and framework for automated class testing. *Testing Object-Oriented Software*, Kung DC, Hsia P, Gao J (eds.). IEEE Computer Society: Silver Spring, MD, 1998; 152–176.
45. Aichernig BK, Gerstinger A, Aster R. Formal specification techniques as a catalyst in validation. *Proceedings of the 5th International High Assurance Systems Engineering Symposium*, Srimani P (ed.). IEEE Computer Society: Silver Spring, MD, 2000; 203–206.
46. Khurshid S, Marinov D. TestEra: Specification-based testing of Java programs using SAT. *Automated Software Engineering Journal* 2004; **11**(4):403–434.



DOI: 10.1002/stvr