Unit Testing of Z Specifications

Mark Utting[1] and Petra Malik[2]

[1] Department of Computer Science, The University of Waikato, NZ
marku@cs.waikato.ac.nz,
[2] Faculty of Engineering, Victoria University of Wellington, NZ
petra.malik@mcs.vuw.ac.nz

Abstract. We propose a simple framework for validation unit testing of Z specifications, and illustrate this framework by testing the first few levels of a POSIX specification. The tests are written in standard Z, and are executable by the CZT animator, ZLive.

1 Introduction

In [Hoa03], Hoare proposes a grand challenge for computer science—a verifying compiler—and inspired researchers from all over the world to work jointly towards this ambitious goal. A key part of the grand challenge is a series of case studies [BHW06], which provide examples of specified and verified code and can be used as benchmarks to exercise and test current and future verification tools.

The Mondex case study was tackled as a first pilot case study in 2006. Several teams using a variety of techniques and tools worked on a fully automated proof of the Mondex smart-card banking application. A verifiable filesystem has been proposed [JH07] as another mini challenge. An initial small subset of POSIX has been chosen [FFW07] and participants of the ABZ 2008 conference were challenged to contribute to this project.

Lots of work and effort is typically put into either deriving or verifying a correct lower level specification or implementation from an abstract specification. We argue that before those activities, the abstract specification should be carefully validated against the requirements. While there is hope that verification can be mostly automated, validation remains a task for human designers.

This paper proposes a simple approach to validate Z specifications by writing positive and negative unit tests for each schema within the specification. These tests give the designer more confidence that their specification reflects their intentions, allows regression testing after each modification of the specification, and can help the unfamiliar reader of the specification to understand the specification more quickly and easily. The framework has been used to design tests for the first few levels of a refactored version of Morgan and Suffrin's Z specification of the POSIX file system [MS84]. The tests were executed and checked by the CZT animator ZLive.

The structure of this paper is as follows. In Section 2, we present our testing framework. Section 3 gives an overview of the ZLive animator, which is used to evaluate the tests. Section 4 describes the refactored POSIX specification.

In Section 5, the tests for the data system are explained and Section 6 shows how they can be promoted to test the storage system. Finally, Section 7 gives conclusions.

The main contributions of the paper are: the unit testing framework for validating Z specifications, a simple style for promoting tests from one level to another, the refactored and standard-compliant POSIX specification, the example unit tests we have developed for it, and the use of the ZLive animator to execute those tests.

2 How to Test a Z Specification

This section introduces a simple framework for expressing unit tests of a Z specification. We assume that the specification is written in the common Z *sequential* style, with each section containing a state schema, an initialisation schema, several operation schemas, as well as various auxiliary definitions and schemas.

The first question that must be asked is why we do not use an existing testing framework for Z, such as the Test Template Framework (TTF) [Sto93,CS94] or Heirons' Z-to-FSM approach [Hie97]. The main difference is that they all aim at generating tests *from* a Z specification, in order to test some implementation of that specification. But in this paper, we want to design some tests that actually test the Z specification itself. So the goal of those approaches is *verification* (of some implementation), whereas our goal is *validation* (of the Z specification). This results in quite a different style of testing.

An important philosophical difference is that when our aim is the *validation* of a Z specification, the correctness of our tests must be ensured by some external means (not just the specification itself), to avoid circular reasoning. That is, we cannot generate tests from the specification, then test them against that same specification, and expect to find any errors. There needs to be some independence between the tests and the system being tested. In this paper we avoid such circular reasoning by designing our tests manually, based on the informal English description of the POSIX operations. So the oracle for the correctness of the tests is our human understanding of the POSIX requirements.

A practical difference is that when testing an implementation (for verification testing), we can control its inputs but not its outputs, whereas when testing a Z specification (for validation purposes) we can ask arbitrary questions about inputs or outputs. So when validating a Z specification, we can use a richer variety of tests. In this paper, by a *test* of a Z schema Op we mean a boolean property of some finite/small subset of S that can (in principle) be enumerated in a reasonable time. For example, we may be able to test Op by instantiating it with a specific output value and asking which input values could generate that output value – such 'tests' are not possible on black-box implementations.

Another difference is that when validating a specification, we want to design both positive tests, which test that a given behaviour is allowed by the specification, and negative tests, which test that a given behaviour is not allowed by the specification. In contrast, when testing whether an implementation is

a refinement of a specification, the implementation is usually free to add behaviour outside the precondition of the specification and we can therefore use only positive tests.[3]

To illustrate our testing framework, we shall use a simple example specification given by the following specification, which defines the shape shown in Figure 1. Our tests will be written using standard Z conjecture paragraphs, so that they can be checked by theorem provers or by the ZLive animator. For each section S, we write the unit tests for that section within a separate section (typically called *STests*) that has S as a parent. This clearly identifies the tests and separates them from the rest of the specification, so that tools that operate on the main specification can ignore the unit tests.

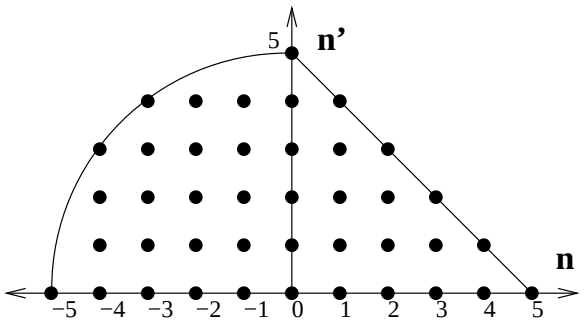

Fig. 1. A graph of the solutions to the Wedge schema

section *wedge* **parents** *standard_toolkit*

$$State == [n : \text{-}\,10 \,..\, 10]$$
$$Wedge == [\Delta State \mid n * n + n' * n' \leq 25;\ n + n' \leq 5;\ 0 \leq n']$$

2.1 Positive Tests

The first kind of testing we want to do is positive tests to check that an operation has a desired input-output behaviour. When specifying tests, it is usual to specify the expected output values as well as the input values, so we write all the inputs and output values as a Z binding and use a membership test. Here are two examples that test the non-deterministic output behaviour of Wedge when the input n is zero.

[3] Note that a test of exceptional behaviour that is formalised in the specification is a positive test.

section *wedgeTest* **parents** *wedge*

theorem Wedge0Gives0
$\vdash? \langle n == 0, n' == 0 \rangle \in Wedge$

theorem Wedge0Gives5
$\vdash? \langle n == 0, n' == 5 \rangle \in Wedge$

Note that since 2007, the Z standard requires conjectures to be written within a LaTeX *theorem* environment, and allows each conjecture to be given a name. Our naming convention for tests is that the name of a test should start with the name of the operation being tested, followed by some phrase that expresses the essential property of the test.

An alternative style of writing a suite of positive tests is to name each test tuple, group the tuples into a set, and then test them using a single subset conjecture.

$Wedge0Gives0 == \langle n == 0, n' == 0 \rangle$
$Wedge0Gives5 == \langle n == 0, n' == 5 \rangle$

theorem WedgePos
$\vdash? \{ Wedge0Gives0, Wedge0Gives5 \} \subseteq Wedge$

2.2 Negative Tests

It is also useful to perform *negative* tests to validate an operation. For example, we may want to check that an input value is outside the precondition of the operation, or check that a certain output can never be produced by the operation. The idea is to validate the specification by showing that its behaviour is squeezed in between the set of positive tests and the set of negative tests. The more positive and negative tests that we design, the more sure we can be that we have specified the desired behaviour, rather than allowing too many or too few behaviours.

We can write negative tests using a negated membership test ($test \notin Op$) or, equivalently, we can test that the tuple is a member of the negated schema ($test \in \neg\, Op$). Our conjecture naming convention is the same as for positive tests, but the phrase after the operation name usually contains a negative word such as *not* or *cannot* to emphasize that it is a negative test.

theorem Wedge0NotNeg
$\vdash? \langle n == 0, n' == -1 \rangle \notin Wedge$

theorem Wedge0Not6
$\vdash? \langle n == 0, n' == 6 \rangle \in \neg\ Wedge$

We can also test just the precondition of the operation.

theorem WedgePreNot6
$\vdash? \langle n == 6 \rangle \notin \mathbf{pre}\ Wedge$

As we did for positive tests, we may also combine several negative tests into a group. In this case, our conjecture may be written as $tests \subseteq \neg\ Op$, or equivalently we may check that the intersection of the negative test suite and the operation is empty, $tests \land Op = \varnothing$. The latter style is often more convenient, since it allows us to write *tests* using schema notation, and to omit variables that we are not interested in (because the schema conjunction will expand the type of *tests* to match the type of *Op*). For example, the following two negative test suites check that 6 can never be an input for *Wedge*, and that 6 can never be an output of *Wedge*.

theorem Wedge6Not
$\vdash? ([n == 6] \land Wedge) = \varnothing$

theorem WedgeNot6
$\vdash? ([n' == 6] \land Wedge) = \varnothing$

It is sometimes convenient to write these kinds of negative test values *within* the operation schema (e.g., $[Wedge \mid n = 6] = \varnothing$), which can make the tests even more concise.

2.3 Promoting Tests

A heavily-used pattern in the POSIX Z specification is the use of *promotion* to lift the operations of one data type up to work on a more complex data type. It is useful to be able to promote the tests of those operations as well. To illustrate an elegant way of doing this, we shall promote the *Wedge* tests up to the following function space:

section *manyStates* **parents** *wedge*

$ManyStates == [states : \mathbb{N} \to \mathbb{Z}]$

ΦManyStates

$\Delta State$
$\Delta ManyStates$
$curr? : \mathbb{N}$

$(curr? \mapsto n) \in states$
$states' = states \oplus \{curr? \mapsto n'\}$

We promote the *Wedge* operation up to the *ManyStates* level simply by conjoining *Wedge* with the *framing schema* $\Phi ManyStates$. The effect is to apply the *Wedge* operation to just the *curr*? element of the *states* function.

$$ManyWedge == \Phi ManyStates \land Wedge$$

We use the same framing schema to promote the tests. But to ensure that all inputs of the promoted operation are given, we find it useful to further instantiate the framing schema $\Phi ManyStates$, to obtain a testing-oriented framing schema ($\Phi ManyStatesTest$) that also specifies which *curr*? input will be tested and an initial value for the *states* mapping.

section *manyStatesTest* **parents** *manyStates*, *wedgeTest*

$$\Phi ManyStatesTest == [\Phi ManyStates;\ curr? == 1 \mid states = \{0 \mapsto 3, 1 \mapsto n\}]$$

theorem ManyWedgePos
$$\vdash? (\{Wedge0Gives0, Wedge0Gives5\} \land \Phi ManyStatesTest) \subseteq ManyWedge$$

In fact, this style of promoted test theorem will always be true, because for any test suite *OpTests* of an operation *Op*, and any promoted operation defined as $Op_P == Op \land \Phi P$, it is true that:

$$(OpTests \subseteq Op) \land (\Phi PTests \subseteq \Phi P) \Rightarrow (OpTests \land \Phi PTests \subseteq Op_P)$$

Proof: follows from the monotonicity of $\subseteq$ and schema conjunction. □

However, there is a possibility that some of the promoted tests might be inconsistent with the framing schema (either ΦP or $\Phi PTests$), which would mean that the set $OpTests \land \Phi PTests$ would be empty or smaller than our original set of tests *Optests*. If we want to check that all of the original tests can be promoted without being lost, we can check the conjecture $(\Phi PTests \upharpoonright OpTests) = Optests$.

If this mass-promotion test fails, we may want to check each promoted test vector $v \in Optests$ separately. To do this, we can define the promoted vector as $pv == (\mu \Phi PTests \land \{v\})$, and then we can test $pv \in Op_P$.

3 The ZLive Animator

One of the tools available in the CZT system is the ZLive animator. It is the successor to the Jaza animator for Z [Utt00], and its command line interface is largely backwards compatible with that of Jaza. However, the animation algorithm of ZLive is quite different and more general. It is based on an extension of the *Z mode* system developed at Melbourne University by Winikoff and others [KWD98,Win98].

When an expression or predicate is evaluated in ZLive, it is parsed and type-checked, and then unfolded and simplified using the *transformation rules* system

of CZT [UM07]. This unfolds schema operators, expands definitions and performs a variety of simplifications. For example, the test $\langle\!| n == 0, n' == 0 |\!\rangle \in \mathit{Wedge}$ is unfolded to:

$$\langle\!| n == 0, n' == 0 |\!\rangle \in [n : \text{-}10 \mathinner{..} 10;\ n' : \text{-}10 \mathinner{..} 10 \mid n * n + n' * n' \leq 25;\ n + n' \leq 5;\ 0 \leq n']$$

The resulting term is then translated into a sequence of primitive relations, each of which corresponds to a single Z operator. For example, the term $n * n + n' * n' \leq 25$ is translated into a sequence of four relations:

$$\begin{array}{ll} \mathit{FlatMult}(n, n, \mathit{tmp}1), & // \ n * n = \mathit{tmp}1 \\ \mathit{FlatMult}(n', n', \mathit{tmp}2), & // \ n' * n' = \mathit{tmp}2 \\ \mathit{FlatPlus}(\mathit{tmp}1, \mathit{tmp}2, \mathit{tmp}3), & // \ \mathit{tmp}1 + \mathit{tmp}2 = \mathit{tmp}3 \\ \mathit{FlatLessThanEq}(\mathit{tmp}3, 25) & // \ \mathit{tmp}3 \leq 25 \end{array}$$

ZLive next performs a bounds analysis phase, which does a fixpoint calculation to infer conservative lower and upper bounds for each integer variable, bounds on the size and contents of sets, and aliasing between variables. This infers that $n \in \text{-}10 \mathinner{..} 5$ and $n' \in 0 \mathinner{..} 10$.

The last analysis phase attempts to reorder the sequence of primitive *Flat* ... relations into an efficient computation order. Each of the relations can be executed in one or more *modes*. A mode determines whether each parameter is an input or an output. In addition, ZLive estimates the expected number of results for each mode. For example, the mode IIO:1 for $\mathit{FlatPlus}(x, y, z)$ means that x and y are inputs and z is an output (with one result expected), while mode III:0.6 means they are all inputs and there is a 60% probability of getting a result. The reordering algorithm gives preference to modes that have a small number of expected results, which means that filter predicates are moved as near to the front as possible, which reduces the search space.

Finally, ZLive enumerates all possible results via a depth-first backtracking search of all the solutions generated by the sorted sequence. However, for membership tests such as $A \in S$, where A is a known value and S is a set comprehension, it substitutes the value of A into the set comprehension for S and then checks whether the set is empty or not – this avoids generating all of S.

ZLive can usually evaluate expressions that range over finite sets only, and can sometimes handle expressions that contain a mixture of finite and infinite sets. So it is a useful tool for checking the correctness of tests, and can sometimes be used to generate one or all solutions of a partially specified test or schema.

4 POSIX Standardized

In this section, we briefly describe the refactored POSIX specification. The main change was to break up the original specification into sections. Figure 2 shows the structure of the resulting Z sections, using a notation similar to a UML class diagram. Each box represents a Z section, and the three parts within each box

show the name of the section, the main variables within the state schema of that definition, and the names of its operation schemas (we omit Init schemas and auxiliary schemas). We also added a state schema and initialization schema to some of the sections (e.g., the *ds* section) and made several naming changes so that the specification follows the usual Z sequential style more closely.

section *ds* **parents** *standard_toolkit*

This section specifies the data system (ds) of the filing system.

$$BYTE == 0 \,..\, 255$$
$$ZERO == 0$$
$$FILE == \mathrm{seq}\, BYTE$$
$$DS == [file : FILE]$$
$$InitDS == [DS' \mid file' = \langle\rangle]$$

The *after* operator returns the subfile that starts after a given offset. We write this as an explicit definition (==) rather than axiomatically, because it is clearer, avoids possible inconsistency, and is easier to evaluate.

function 42 **leftassoc** (*_after_*)

$$\_after\_ == (\lambda f : FILE;\ offset : \mathbb{N} \bullet (\lambda i : 1..\#f - offset \bullet f(i + offset)))$$

The *readFile* operation is defined similar to the one in Morgan and Sufrin's specification but we use the usual Ξ notation for convenience here.

readFILE

ΞDS
$offset?, length? : \mathbb{N}$
$data! : FILE$

$data! = (1 \,..\, length?) \lhd (file\ after\ offset?)$

The auxiliary function *zero* returns a *FILE* containing a given number of *ZERO* bytes. The infix operator *shift* takes a *FILE* and an offset and shifts the content of the file by the offset. Once again, we give an explicit rather than an axiomatic definition.

$$zero == (\lambda n : \mathbb{N} \bullet (\lambda k : 1 \,..\, n \bullet ZERO))$$

function 42 **leftassoc** (*_shift_*)

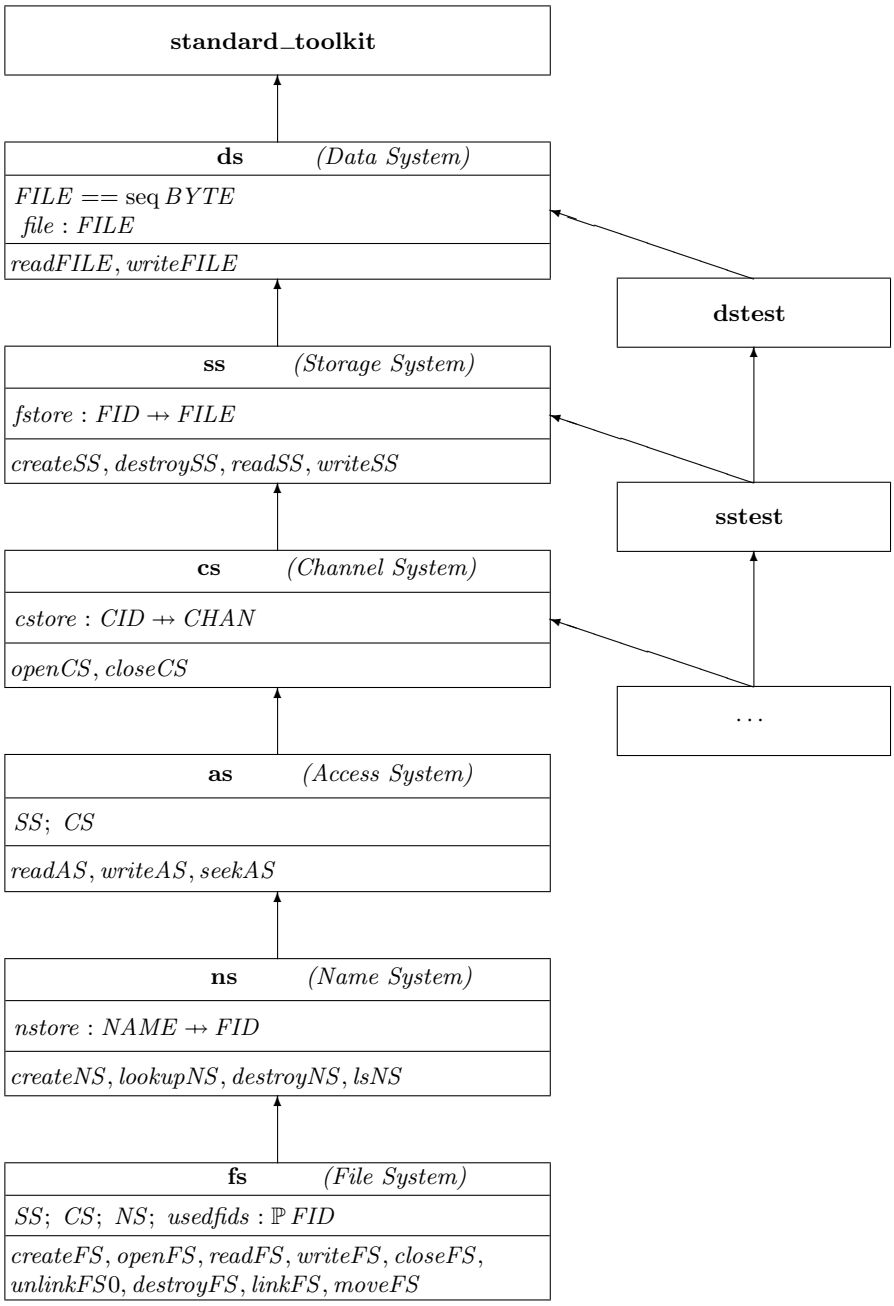

Fig. 2. Overview of Z Sections in the Refactored POSIX Specification

$$\_shift\_ == (\lambda f : FILE;\ offset : \mathbb{N} \bullet (1 \mathinner{..} offset) \lhd (zero\ offset \frown f))$$

While the *readFILE* operation does not change the file, the *writeFile* operation given next changes the file. It is defined similar to the *writeFile* operation given in Morgan and Sufrin's specification but we use the usual Δ notation for convenience here.

writeFILE

$$\Delta DS$$
$$offset? : \mathbb{N}$$
$$data? : FILE$$
$$file' = (zero\ offset? \oplus file) \oplus (data?\ shift\ offset?)$$

5 Testing the DS Specification

section *dstest* **parents** *ds*

We start by testing the *InitDS* schema.

theorem InitDSEmpty
$\vdash? \langle\!| file' == \langle\rangle |\!\rangle \in InitDS$

theorem InitDSNot3
$\vdash? \langle\!| file' == \langle 3 \rangle |\!\rangle \notin InitDS$

For *readFILE*, we design a set of positive tests that all work on the same input file contents, $eg1$. So when writing the set of tests, it is convenient to use the schema calculus to factor out the unchanging $file, file'$ from the other test values.

$$eg1 == \langle 1, 255 \rangle$$
$$dsPos == [file == eg1;\ file' == eg1] \land \{\langle\!| offset? == 0, length? == 0, data! == \langle\rangle |\!\rangle, \langle\!| offset? == 0, length? == 3, data! == \langle 1, 255 \rangle |\!\rangle, \langle\!| offset? == 1, length? == 1, data! == \langle 1 \rangle |\!\rangle, \langle\!| offset? == 3, length? == 2, data! == \langle 0, 0 \rangle |\!\rangle \}$$

theorem readFILEPos
$\vdash? dsPos \subseteq readFILE$

Here is the output from ZLive when we use its `conjectures` command to evaluate all the conjectures in this *dstest* section.[4]

```
dstest> conjectures
Conjecture on line 7 (InitDSEmpty)
true
Conjecture on line 11 (InitDSNot3)
true
Conjecture on line 29 (readFILEPos)
false
```

To investigate the failing conjecture *readFILEPos* on line 29, we ask ZLive to evaluate $dsPos \setminus readFILE$. This displays just the test vectors that are not members of *readFILE*, which tells us that the third and fourth tests failed. For each of these, we investigate why it failed by using the ZLive '`do`' command to search for any solution to the *readFILE* schema with the given input values. For the third test, we get this output:

```
dstest> do [readFILE | file=eg1; offset?=1; length?=1]
```

$$1 : \langle\!| file == \{(1,1),(2,255)\}, file' == \{(1,1),(2,255)\}, \\ offset? == 1, length? == 1, data! == \{(1,255)\} |\!\rangle$$

Oops, this test should have had 255 in *data*! rather than 1, because in POSIX, $offset? = 1$ refers to the *second* byte of the file.

```
dstest> do [readFILE | file=eg1; offset?=3; length?=2]
```

$$1 : \langle\!| file == \{(1,1),(2,255)\}, file' == \{(1,1),(2,255)\}, \\ offset? == 3, length? == 2, data! == \{\} |\!\rangle$$

Ah, of course! Reading past the end of the file should return empty *data*!, rather than zeroes. (When designing this fourth test, Mark was incorrectly thinking of the behaviour of the Write command past the end of the file, which inserts zeroes automatically.) Once these two errors in the expected output values are corrected, all tests give true.

5.1 Negative Tests for *readFILE*

Our first two negative tests check that -1 is not a valid input for *offset*? or *length*?. In the latter test (*ReadNotLenNeg*), we show how we can write the test values inside the *readFILE* schema, which can sometimes be more convenient.

[4] We have added the conjecture names into the ZLive output by hand in this example, but hope to automate this in the future. The problem is that the Z standard currently does not pass the conjecture names from the LaTeX markup to the Unicode markup, so getting access to the names within the parser and ZLive will require some extensions to the Z standard, which we have not yet made.

theorem ReadNotOffNeg
$\vdash? ([offset? == -1] \land readFILE) = \{\}$

theorem ReadNotLenNeg
$\vdash? [readFILE \mid length? = -1] = \{\}$

The remaining negative tests are partially specified, so each conjecture actually checks a set of negative test tuples. The *ReadNoChange* test checks that the read operation does not change the contents of our example file $eg1$. The *ReadNotLonger* test checks that the output $data!$ is never longer than the contents of file $eg1$. This is actually proving that the property $\# data! > 2$ is false for all lengths $0 \,..\, 3$, which is a form of finite proof by exhaustive enumeration.

theorem ReadNoChange
$\vdash? [readFILE \mid file = eg1;\ file' \neq eg1] = \{\}$

theorem ReadNotLonger
$\vdash? [readFILE \mid file = eg1;\ offset? = 0;\ length? < 4;\ \# data! > 2] = \{\}$

This illustrates that there is a continuum between testing and proof. We usually test just one input-output tuple at a time, but the idea of testing can be extended (as in this paper) to allow a given property to be evaluated for all members of a finite/small set. This is similar to model-checking, where properties of finite systems are proved by exhaustive enumeration. Animators like ZLive use a mixture of symbolic manipulation techniques and exhaustive enumeration. The more they use symbolic manipulation, the closer they become to general theorem provers. So the testing-proof continuum ranges from testing of single input-output tuples, through enumeration (or model-checking) of finite systems, to full symbolic proof.

We can write positive and negative unit tests for the *writeFile* operation in a similar way to *readFile*, but space does not permit us to show the details of this.

6 Testing the SS Specification

The storage system is reponsible for mapping *file identifiers FID* to file contents. While testing, we instantiate FID to naturals, so test values are easier to write.

section *ss* **parents** *ds*

$FID == \mathbb{N}$
$SS == [fstore : FID \pfun FILE]$

The *ss* section then defines *createSS* and *destroySS* operations, plus the following framing schema, which is used to promote *readFILE* and *writeFILE*.

$$
\begin{array}{|l}
\hline
\Phi SS \\
\Delta SS;\ \Delta DS;\ fid? : FID \\
\hline
fid? \in \mathrm{dom} fstore \\
file = fstore(fid?) \\
fstore' = fstore \oplus \{fid? \mapsto file'\} \\
\hline
\end{array}
$$

$$readSS == (\Phi SS \land readFILE) \setminus (file, file')$$

section *sstest* **parents** *dstest*, *ss*

To promote the ds tests, we define a special case of the ΦSS framing schema.

$$\Phi SSTest == [\Phi SS \mid fid? = 101;\ fstore = \{100 \mapsto \langle 3, 5\rangle, 101 \mapsto file\}]$$

Then we can promote the *dsPos* tests and check if they satisfy *readSS*.

theorem SSTestPos

$$\vdash? (dsPos \land \Phi SSTest) \setminus (file, file') \subseteq readSS$$

Unfortunately, due to an inefficiency in its sorting/optimization algorithms, ZLive currently says the left side of the $\subseteq$ is too large to evaluate. However, it should be capable of evaluating it, and we expect that it will be able to in the next few months.

7 Conclusions

In this paper, we have proposed a framework for unit testing Z specifications. The framework uses the sections and conjectures of the Z standard to allow various kinds of validation tests to be expressed in an elegant and concise style. The ability to promote a large set of tests in a single expression makes it practical to develop multiple layers of tests, matching the layers of the specification. Testing is a useful validation technique for specifications, especially when the execution of the tests can be automated, as we have done with ZLive. We believe that most Z specifications should include validation unit tests in this style.

We plan to add a `unittest` command to ZLive that executes all the unit tests in all the sections whose names end with 'Test'. This will make it easy to rerun all unit tests after each modification of a specification, so will support regression testing during development of Z specifications. It would also be useful

if ZLive measured structural coverage metrics of the operation schemas during testing, so that we can see what level of, say, branch coverage (each predicate evaluating to true and to false) our test suite obtains.

Our style of unit testing is quite complementary to the specification validation facilities of ProB/ProZ [LB08], because we focus on unit testing of each individual operation schema using a manually designed test suite, while ProB focusses on automatic testing of sequences of operations, and tests only a few input values of each operation. The goal of our unit testing is to test the input-output functionality of each operation, while the main goal of ProB is to try to validate several standard system properties such as absence of deadlock and preservation of invariants.

The refactored Z specification of POSIX, and our simple test suite, may be a useful starting point for other researchers who want to work on refinement or proofs about the POSIX case study. It is available from the CZT sourceforge website [CZT]. Interestingly, the original POSIX specification did include several examples that used test values to illustrate some of the operations, but those examples were written within the English commentary, so were not even type-checked, let alone proved correct. Our unit tests are more systematic and are formalized so that they can be checked automatically by an animator like ZLive.

References

[BHW06] J. C. Bicarregui, C. A. R. Hoare, and J. C. P. Woodcock. The verified software repository: a step towards the verifying compiler. *Formal Aspects of Computing*, 18(2):143–151, 2006.

[CS94] D. A. Carrington and P. Stocks. A tale of two paradigms: formal methods and software testing. In *Proceedings of the 8th Z User Meeting (ZUM'94)*, pages 51–68, Cambridge, June 1994. Springer Verlag.

[CZT] Community Z tools. `http://czt.sourceforge.net`.

[FFW07] Leo Freitas, Zheng Fu, and Jim Woodcock. POSIX file store in Z/EVES: an experiment in the verified software repository. In *ICECCS '07: Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems*, pages 3–14. IEEE Computer Society, 2007.

[Hie97] R. Hierons. Testing from a Z specification. *Software Testing, Verification & Reliability*, 7:19–33, 1997.

[Hoa03] Tony Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, 50(1):63–69, 2003.

[JH07] Rajeev Joshi and Gerard J. Holzmann. A mini challenge: build a verifiable filesystem. *Formal Aspects of Computing*, 19(2):269–272, 2007.

[KWD98] Ed. Kazmierczak, Michael Winikoff, and Philip Dart. Verifying model oriented specifications through animation. In *Proceedings of the 5th Asia-Pacific Software Engineering Conference*, pages 254–261. IEEE Computer Society Press, December 1998.

[LB08] Michael Leuschel and Michael Butler. ProB: An Automated Analysis Toolset for the B Method. *Journal Software Tools for Technology Transfer*, page Accepted for publication, 2008.

[MS84] Carroll Morgan and Bernard Sufrin. Specification of the UNIX filing system. *IEEE Transactions on Software Engineering*, 1984.

[Sto93] P. Stocks. *Applying Formal Methods to Software Testing*. PhD thesis, The University of Queensland, 1993. Available from `http://www.bond.edu.au/it/staff/publications/PhilS-pubs.htm`.

[UM07] Mark Utting and Petra Malik. Transforming Z with rules. In *ZUM07 at ICECCS07 in conjunction with FMinNZ07*, 2007. Auckland, July 2007.

[Utt00] Mark Utting. Data structures for Z testing tools. In G. Schellhorn and W. Reif, editors, *FM-TOOLS 2000, 4th Workshop on Tools for System Design and Verification*, volume 2000-07. Ulmer Informatik Berichte, May 2000.

[Win98] Michael Winikoff. Analysing modes and subtypes in Z specifications. Technical Report 98/2, Melbourne University, 1998.