

# The Fastest 1.7 User's Guide

## Automating Software Testing

Maximiliano Cristiá  
`cristia@cifasis-conicet.gov.ar`

UNR and CIFASIS  
Rosario – Argentina

Release date: March 2021

# Contents

<b>1</b>	<b>Introduction to Model-Based Testing</b>	<b>4</b>
1.1	Software Testing . . . . .	4
1.2	Functional Correctness and Formal Specifications . . . . .	4
1.3	Model-Based Testing . . . . .	5
<b>2</b>	<b>The Test Template Framework</b>	<b>7</b>
2.1	TTF Key Concepts . . . . .	8
2.1.1	Input Space . . . . .	8
2.1.2	Valid Input Space . . . . .	8
2.1.3	Test Class . . . . .	8
2.1.4	Testing Tactic . . . . .	8
2.1.5	Testing Tree . . . . .	9
2.1.6	Pruning Testing Trees . . . . .	9
2.1.7	Abstract Test Case . . . . .	10
<b>3</b>	<b>Running an Example on Fastest</b>	<b>11</b>
3.1	The Z Specification . . . . .	11
3.2	Applying Fastest to <i>Withdraw</i> . . . . .	13
<b>4</b>	<b>Tips on Writing Z Models for Fastest</b>	<b>16</b>
4.1	The Z Notation Is Not Fully Supported . . . . .	16
4.2	Fastest Conforms to the Z ISO Standard . . . . .	16
4.3	Fastest Is Meant to be Used for Unit Testing . . . . .	16
4.4	Be Careful in Testing Compound Operations . . . . .	17
4.5	Do Not Include the State Invariant in the State Schema . . . . .	18
4.6	Keep the State Schema Focused on a Set of Related Operations . . . . .	18
4.7	Avoid Using Quantifiers . . . . .	19
4.8	Avoid Axiomatic Definitions . . . . .	19
4.9	Avoid Arbitrary Numeric Constants . . . . .	20
4.10	Do Not Use Total Functions Over Infinite Sets . . . . .	21
<b>5</b>	<b>User's Manual</b>	<b>22</b>
5.1	Installing and Executing Fastest . . . . .	22
5.1.1	Running Fastest and Entering Commands . . . . .	22
5.2	Steps of a Testing Campaign . . . . .	23
5.3	Loading a Specification and Selecting Schemas . . . . .	24
5.4	Dealing with Axiomatic Definitions . . . . .	24
5.4.1	Basic Types . . . . .	25
5.4.2	Symbolic Constants . . . . .	25
5.4.3	Equalities . . . . .	26
5.4.4	Equivalences . . . . .	26
5.4.5	All Other Declarations . . . . .	26
5.4.6	Command <code>setaxdef</code> . . . . .	26
5.5	Applying Testing Tactics and Generating Testing Trees . . . . .	28
5.5.1	Disjunctive Normal Form . . . . .	28
5.5.2	Standard Partition (Fastest's name SP) . . . . .	30
5.5.3	Free Type (Fastest's name FT) . . . . .	30
5.5.4	In Set Extension (Fastest's name ISE) . . . . .	30
5.5.5	Proper Subset of Set Extension (Fastest's name PSSE) . . . . .	30

5.5.6	Subset of Set Extension (Fastest's name SSE)	31
5.5.7	Numeric Ranges (Fastest's name NR)	31
5.5.8	Mandatory Test Set (Fastest's name MTS)	31
5.6	Manually pruning the testing tree	32
5.7	Generating Abstract Test Cases	32
5.8	Timeout for Abstract Test Case Generation	33
5.9	Exploring and Saving the Results	33
5.10	How to Quit Fastest	35
<b>A</b>	<b>Z Features Unsupported by Fastest</b>	<b>36</b>
<b>B</b>	<b>Test Classes Generated for <i>KeepMaxReading</i></b>	<b>37</b>
<b>C</b>	<b>Standard Partitions</b>	<b>39</b>
C.1	Sets	39
C.2	Integers	40
C.3	Relations	41
C.4	Sequences	42

# 1 Introduction to Model-Based Testing

Software construction has proved to be more complex than expected. Most often software projects run beyond budget, are delivered late and having many errors. Only an insignificant portion of the products of the software industry are sold with warranty. There is a number of reasons for this state of the practice, but companies usually complain about the costs of software verification as the cause of not doing it thoroughly [Bro95, page 20] [BCK03, page 88] [Pfl01, page 157] [McC04] [RTI02, table ES-1 at page ES-5]. Reducing the costs of verification would imply more projects within budget and less errors. One of the most promising strategies for reducing the costs of verification is making it as automatic as possible. On the other hand, the software industry relies almost exclusively on testing to perform the functional verification of its products. Currently, testing is essentially a manual activity that automates only the most trivial tasks [MFC<sup>+</sup>09, RTI02].

## 1.1 Software Testing

Software testing can be defined as the dynamic verification of a program by running it on a finite set of carefully chosen test cases, from the usually infinite input domain, and comparing the actual behavior with respect to the expected one [DBA<sup>+</sup>01, UL06]. We want to remark the following:

- Testing implies running the program as opposed to, say, static analysis performed on the source code.
- The set of test cases on which the program will be executed is finite and usually very small, compared with the size of the input domain.
- These test cases must be selected, i.e there are some criteria or rules that must be followed in order to choose test cases. It would be wrong a selection process guided by the mood of the engineer.
- The output produced by the program for each test case must be compared with the expected output. If both agree then the program is correct on that test case; otherwise some error has been found. The artefact that helps to decide the presence of an error is called oracle.

Many qualities of a program can be tested. For example, performance, portability, usability, security and so on. Although they are all important, functional correctness is perhaps the one on which industry pays more attention. In many contexts, for instance, performing poorly is bad but performing wrongly is worse.

Traditionally, the testing process has been divided into five steps as shown in Figure 1. The idea is to start testing small portions of the system under test (SUT) called units—usually they are subroutines, procedures or functions—in such a way that once they have passed all the tests, they are progressively assembled together. As new units are integrated, the resulting modules are tested. Sometimes it is possible to independently test subsystems of the SUT. Finally, the full system is tested by users. In this way, errors are discovered as earlier as possible.

Fastest focuses on improving a particular unit testing method and providing tool support for the selection of functional test cases for it, as we will shortly see.

## 1.2 Functional Correctness and Formal Specifications

The last item above suggests that there must be some way of determining what the expected output of a program is. In other words, there should be a way of determining whether the program is functionally correct or not. The classical definition of functional correctness is: a program is functionally correct if it behaves according to its functional specification [GJM03, page 17]. This means that two

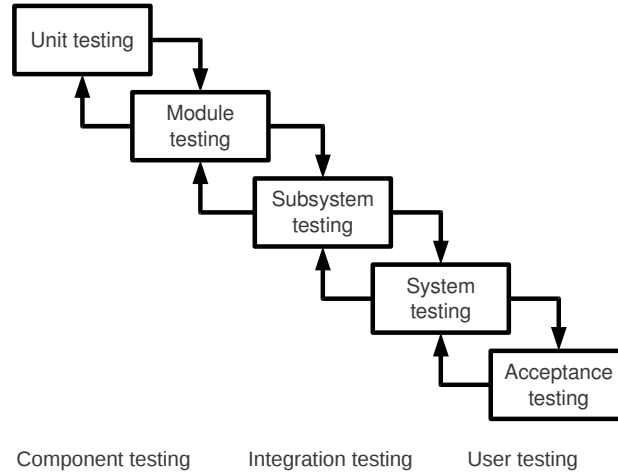


Figure 1: Steps of the testing process

documents or descriptions are needed to perform functional verification: the program itself and its functional specification. In turn, this implies that functional testing is possible only if a specification of the program or SUT is present. The functional specification is sometimes used as the oracle because it is, in fact, the definition of correctness for its implementation.

Furthermore, if automation of the testing process is the goal, then some kind of formal specification is mandatory because otherwise mechanical analysis of the specification becomes unfeasible, turning testing automation unrealistic. A specification is formal if it is written in a formal notation or language [GJM03, page 167]. Formal notations or formalisms for specifying software systems are known as formal methods and have a long and well-established tradition within the Software Engineering community [Bow, HB99].

Fastest focuses on functional testing based on a formal functional specification of the SUT.

### 1.3 Model-Based Testing

When testing and formal specifications are combined we enter into the scope of Model-Based Testing (MBT). MBT is a well-known technique aimed at testing software systems analyzing a formal model or specification of the SUT [UL06, HBB<sup>+</sup>09]. That is, MBT approaches generate test cases from the formal specification of the SUT. The fundamental hypothesis behind MBT is that, as a program is correct if it satisfies its specification, then the specification is an excellent source of test cases.

One of the possible processes of testing a system through a MBT method is depicted in Figure 2. The first step is to analyze the model of the SUT looking for abstract test cases. Usually, MBT methods divides this step into two activities: firstly, test specifications are generated, and, secondly, abstract test cases are derived from them. Although the form of test specifications depends on the particular MBT method, they can be thought as sets of abstract test cases. Test cases produced during the “Generation” step are abstract in the sense that they are written in the same language of the model, making them, in most of the MBT methods, not executable. In effect, during the “Refinement” step these abstract test cases are made executable by a process that can be called *refinement*, *concretization* or *reification*. Note that this not necessarily means that the SUT has been refined from the model; it only says that abstract test cases must be refined. Once test cases have been refined they have to be executed by running the program on each of them. In doing so, the program produces some output for each test case. At this point, some way of using the model as an oracle, to decide whether a given test case has found an error or not, is needed. There are two possibilities depending on the MBT method and the formal notation being used:

1. When the model is analyzed during the “Generation” step, each abstract test case is bound

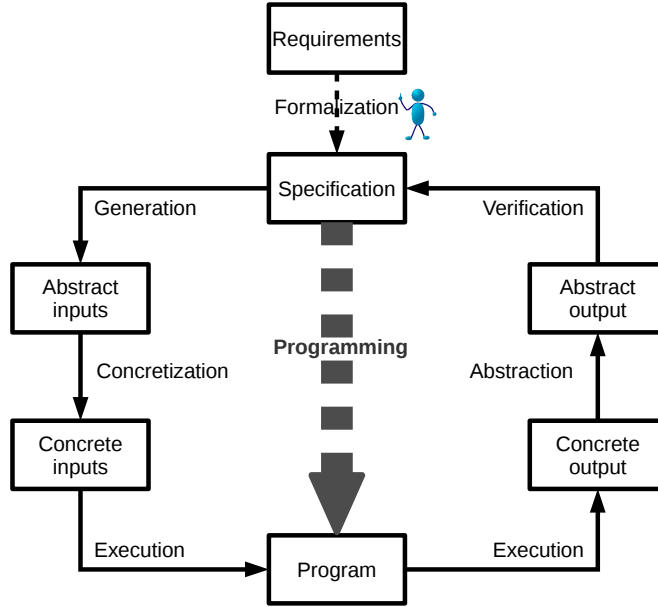


Figure 2: A general description of the MBT process

to its corresponding expected result. Later, these expected results are refined along the same lines of test cases. Finally, the actual output of the program is compared with the result of refining the expected results.

2. The output produced by the SUT for each test case is abstracted at the level of the specification. Then, each abstract test case and its corresponding abstract(ed) output are replaced in the specification. If the specification reduces to *true* then no error was found; if it reduces to *false* then an error was found.

MBT has been applied to models written in different formal notations such as Z [SC96], Finite State Machines (FSM) and their extensions [GGSV02], B [LPU02], algebraic specifications [BGM91], and so on. However, most of the work has focused on the “Generation” step from some variant of FSM for system testing [HBB<sup>+</sup>09, NSV<sup>+</sup>08]. One of the greatest advantages of working with FSM lays in the degree of automation that can be achieved by many MBT methods. On the other hand, FSM pose a strong limit on the kind of systems that can be specified.

Fastest provides support for the “Generation” step from Z specifications as a way to widen the class of systems that can be specified.

## 2 The Test Template Framework

Fastest implements the Test Template Framework (TTF). TTF is a MBT framework proposed by Phil Stocks and David Carrington in [SC96] and [Sto93]. Although the TTF was meant to be notation-independent, the original presentation was made using the Z formal notation. It is one of the few MBT frameworks approaching unit testing.

The TTF deals with the “Generation” step shown in Figure 2. In this framework, each operation within the specification is analyzed to derive or generate abstract test cases. This analysis consists of the following steps, roughly depicted in Figure 3:

1. Define the *input space* (IS) of each operation.
2. Derive the *valid input space* (VIS) from the IS of each operation.
3. Apply one or more *testing tactics*<sup>1</sup>, starting from each VIS, to build a *testing tree* for each operation. Testing trees are populated with nodes called *test classes*.
4. *Prune* each of the resulting testing trees.
5. Find one or more *abstract test cases* from each leaf in each testing tree.

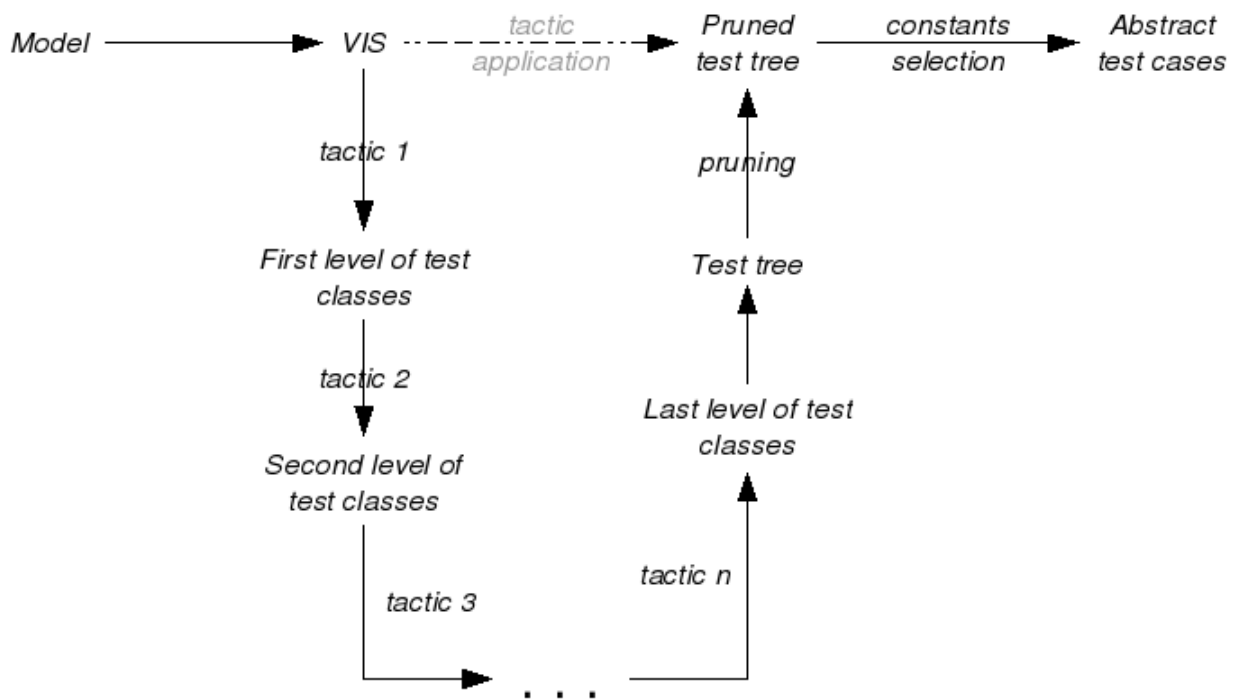


Figure 3: The Test Template Framework process is a detailed view of the “Generation” step shown in Figure 2.

One of the main advantages of the TTF is that all of these concepts are expressed in the same notation of the specification, i.e. the Z notation. Hence, the engineer has to know only one notation to perform the analysis down to the generation of abstract test cases.

The concepts introduced above are explained in the next section. How Fastest implements the TTF, is explained by means of an example in Section 3.

<sup>1</sup>Stocks and Carrington use the term “testing strategies” in [SC96].

- |   |  |
|---|--|
| 1. $S = \emptyset, T = \emptyset$                             | 5. $S \neq \emptyset, T \neq \emptyset, S \subset T$   |
| 2. $S = \emptyset, T \neq \emptyset$                          | 6. $S \neq \emptyset, T \neq \emptyset, T \subset S$   |
| 3. $S \neq \emptyset, T = \emptyset$                          | 7. $S \neq \emptyset, T \neq \emptyset, T = S$   |
| 4. $S \neq \emptyset, T \neq \emptyset, S \cap T = \emptyset$ | 8. $S \neq \emptyset, T \neq \emptyset, S \cap T \neq \emptyset, \neg(S \subseteq T), \neg(T \subseteq S), S \neq T$ |

Figure 4: Standard partition for  $S \cup T$ ,  $S \cap T$  y  $S \setminus T$ .

## 2.1 TTF Key Concepts

In this section the main concepts defined by the TTF are described.

### 2.1.1 Input Space

Let  $Op$  be a Z operation. Let  $x_1 \dots x_n$  be all the input and (non-primed) state variables declared in  $Op$  (or in its included schemas), and  $T_1 \dots T_n$  their corresponding types. The *Input Space* (IS) of  $Op$ , written  $Op^{IS}$ , is the Z schema box defined by  $[x_1 : T_1 \dots x_n : T_n]$ .

### 2.1.2 Valid Input Space

Let  $Op$  be a Z operation. Let  $\text{pre } Op$  be the precondition of  $Op$ . The *Valid Input Space* (VIS) of  $Op$ , written  $Op^{VIS}$ , is the Z schema box defined by  $[Op^{IS} \mid \text{pre } Op]$ .

### 2.1.3 Test Class

Informally, test classes are sets of abstract test cases defined by comprehension; hence each test class is identified by a predicate. Test classes are also called test objectives [UL06], test templates [SC96], test targets and test specifications.

Let  $Op$  be a Z operation and let  $P$  be any predicate depending on one or more of the variables defined in  $Op^{VIS}$ . Then, the Z schema box  $[Op^{VIS} \mid P]$  is a *test class* of  $Op$ . Note that this schema is equivalent to  $[IS_{Op} \mid \text{pre } Op \wedge P]$ . This observation can be generalized by saying that if  $Op^C$  is a test class of  $Op$ , then the Z schema box defined by  $[Op^C \mid P]$  is also a test class of  $Op$ . According to this definition the VIS is also a test class.

If  $Op^C$  is a test class of  $Op$ , then the predicate  $P$  in  $Op^{C'} == [Op^C \mid P]$  is said to be the *characteristic* predicate of  $Op^{C'}$  or  $Op^{C'}$  is *characterized* by  $P$ .

### 2.1.4 Testing Tactic

In the context of the TTF a *testing tactic* is a means of partitioning any test class of any operation. However, some of the testing tactics used in practice actually do not always generate a partition, in the mathematical sense, of some test classes.

For instance, two testing tactics originally proposed for the TTF are the following:

- Disjunctive Normal Form (DNF). By applying this tactic the operation is written in Disjunctive Normal Form and the test class is divided in as many test classes as terms are in the resulting operation's predicate. The predicate added to each new test class is the precondition of one of the terms in the operation's predicate.
- Standard Partitions (SP). This tactic uses a predefined partition of some mathematical operator [Sto93]. For example, the partition shown in Figure 4 is a good partition for expressions of the form  $S \spadesuit T$  where  $\spadesuit$  is one of  $\cup$ ,  $\cap$  and  $\setminus$ .



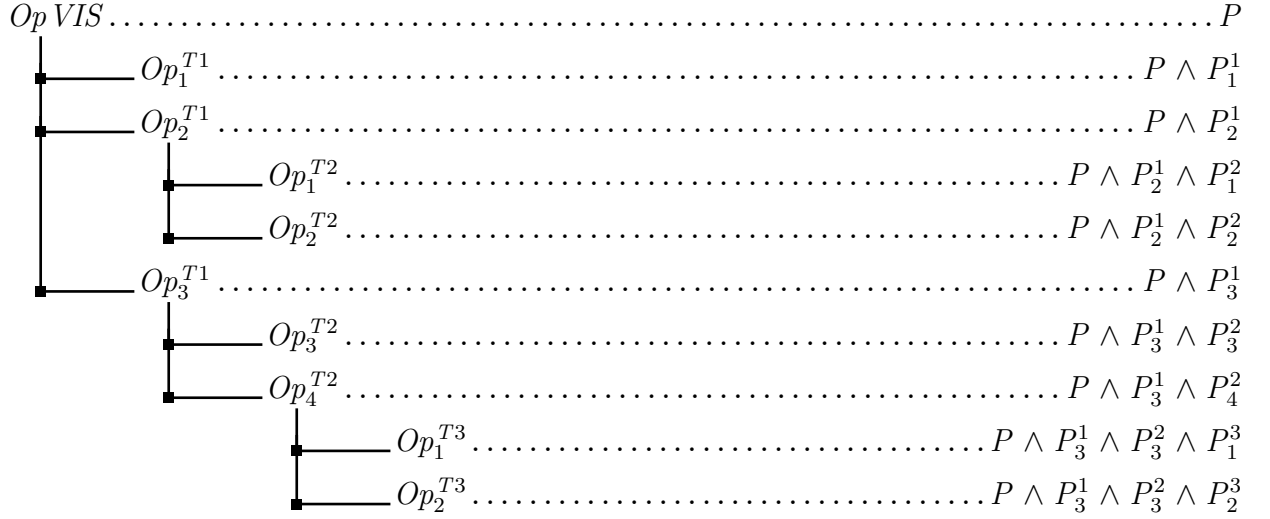


Figure 5: The predicate of a test class at some level is the conjunction of the predicate of its parent test class and its own predicate.

As can be noticed, standard partitions might be changed according to how much testing the engineer wants to perform.

### 2.1.5 Testing Tree

The application of a testing tactic to the VIS generates some test classes. If some of these test classes are further partitioned by applying one or more testing tactics, a new set of test classes is obtained. This process can continue by applying testing tactics to the test classes generated so far. Evidently, the result of this process can be drawn as a tree with the VIS as the root node, the test classes generated by the first testing tactic as its children, and so on. In other words, test classes' predicates obey the relationship depicted in Figure 5. A consequence of this relationship is that the deeper the tree, the more accurate and discovering the test cases. As was noted by Stocks and Carrington in [SC96], the Z notation can be used to build the tree, as follows.

$$\begin{aligned}
Op^{VIS} &== [Op^{IS} \mid P] \\
Op_1^{T1} &== [Op^{VIS} \mid P_1^1] \\
Op_2^{T1} &== [Op^{VIS} \mid P_2^1] \\
Op_1^{T2} &== [Op_2^{T1} \mid P_1^2] \\
Op_2^{T2} &== [Op_1^{T1} \mid P_2^2] \\
Op_3^{T1} &== [Op^{VIS} \mid P_3^1] \\
Op_3^{T2} &== [Op_3^{T1} \mid P_3^2] \\
Op_4^{T2} &== [Op_3^{T1} \mid P_4^2] \\
Op_1^{T3} &== [Op_4^{T2} \mid P_1^3] \\
Op_2^{T3} &== [Op_4^{T2} \mid P_2^3]
\end{aligned}$$

### 2.1.6 Pruning Testing Trees

In general a test class' predicate is a conjunction of two or more predicates. It is likely, then, that some test classes are empty because their predicates are contradictions. These test classes must be pruned from the testing tree because they represent impossible combinations of input values, i.e. no abstract test case can be derived out of them. In this way, pruning the initial testing tree saves CPU time because it avoids searching abstract test cases in empty sets.

### 2.1.7 Abstract Test Case

An abstract test case is an element belonging to a test class. The TTF prescribes that abstract test cases should be derived only from the leaves of the testing tree. Abstract test cases can also be written as Z schema boxes. Let  $Op$  be some operation, let  $Op_{VIS}$  be the VIS of  $Op$ , let  $x_1 : T_1 \dots x_n : T_n$  be all the variables declared in  $Op_{VIS}$ , let  $Op^C$  be a (leaf) test class of the testing tree associated to  $Op$ , let  $P_1 \dots P_m$  be the characteristic predicates of each test class from  $Op^C$  up to  $Op_{VIS}$  (by following the edges from child to parent), and let  $v_1 : T_1 \dots v_n : T_n$  be  $n$  constant values satisfying  $P_1 \wedge \dots \wedge P_m$ . Then, an abstract test case of  $Op^C$  is the Z schema box defined by  $[Op^C \mid x_1 = v_1 \wedge \dots \wedge x_n = v_n]$ .

### 3 Running an Example on Fastest

The purpose of this section is to show and explain the list of commands that should be issued to produce abstract test cases with Fastest. Our intention is to recreate the real work of an engineer using the tool, to show how automatic the work can be. We will give many details but for a thorough explanation read the rest of this user's guide.

Within this section we will assume the following:

- The Z specification was already written and has no syntactic or type errors.
- The engineer is fluent in the TTF and Fastest.
- There are some standard partitions already defined.

The example consists in applying fastest to the *Withdraw* operation of the following Z specification.

#### 3.1 The Z Specification

Think in the savings accounts of a bank. Each account is identified by a so-called account number. Clients can share an account and each client can own many accounts—some of which might be shared with other clients, and some not. The bank requires to keep record of just the balance of each account, and the ID and name of each client. Any person can open an account in the bank becoming its first owner. Owners can withdraw money from their accounts.

As we have said, each savings account is identified by an account number. We need a way to name these account numbers. Since account numbers are used just as identifiers we can abstract them away, not caring about their internal structure. Z provides so-called basic or given types for these cases. The Z syntax for introducing a basic type is:

$[ACCNUM]$

Along the same lines, we introduce basic types for the ID's of clients and their names:

$[UID, NAME]$

We represent the money that clients can deposit and withdraw and the balance of savings accounts as natural numbers. We think that specifying them as real numbers does not add any significant detail to the model, but makes it truly complicated since Z does not provides a native type for real numbers. Then, we define:

$MONEY == \mathbb{N}$

$BALANCE == \mathbb{N}$

The state space is defined as follows:

*Bank*

$clients : UID \rightarrow NAME$

$balances : ACCNUM \rightarrow BALANCE$

$owners : UID \leftrightarrow ACCNUM$

In this way, *clients* is a partial function from *UID* onto *NAME*. It makes sense to define such a function because each person has a unique *UID* but not a unique name; and it makes sense to make

*clients* partial because not every person is a client of the bank all the time. The same is valid for *balances*: there is a functional relationship between account numbers and balances, and not all the account numbers are used all the time in the bank. The symbol  $\leftrightarrow$  defines binary relations. It is correct to define *owners* as a relation, and not as a function, because a given client may own more than one account and each savings account may be owned by many clients.

Now, we can define the initial state of the system as follows:

<i>InitBank</i>	
<i>Bank</i>	
<i>clients</i> = $\emptyset$	
<i>balances</i> = $\emptyset$	
<i>owners</i> = $\emptyset$	

The specification of withdrawing money must say that only account owners can withdraw money from their accounts and they cannot withdraw more money than is available in the account. Hence, the operation requires the ID of the person willing to withdraw money, the account identifier and the amount of money to be withdrawn.

<i>WithdrawOk</i>	
$\Delta Bank$	
$u? : UID$	
$n? : ACCNUM$	
$m? : MONEY$	
$u? \mapsto n? \in owners$	
$n? \in \text{dom } balances$	
$m? > 0$	
$m? \leq balances\ n?$	
$balances' = balances \oplus \{n? \mapsto balances\ n? - m?\}$	
$clients' = clients$	
$owners' = owners$	

Then, we define an “error” schema for the negation of each precondition of the successful case.

*AccountNotExists* ==  
 $[\exists Bank; n? : ACCNUM \mid n? \notin \text{dom } balances]$   
*IncorrectAmount* ==  $[\exists Bank; m? : MONEY \mid m? \leq 0]$   
*NotAnOwner* ==  
 $[\exists Bank; u? : UID; n? : ACCNUM \mid u? \mapsto n? \notin owners]$   
*InsufficientFunds* ==  
 $[\exists Bank; u? : UID; n? : ACCNUM; m? : MONEY \mid$   
 $m? > balances\ n?]$

Therefore, the total operation is defined as follows.

*Withdraw* ==  
 $WithdrawOk$   
 $\vee AccountNotExists$   
 $\vee IncorrectAmount$   
 $\vee NotAnOwner$   
 $\vee InsufficientFunds$

## 3.2 Applying Fastest to *Withdraw*

We are going to generate test cases for the *Withdraw* operation defined above. Fastest is executed from a command-line (or terminal) as follows:

```
java -jar fastest.jar
```

Fastest prints the following prompt from which users can issue commands.

```
Fastest>
```

To enter a command just type-in it along with its arguments and then press the return key. The first step is to load the Z specification. A specification is loaded with `loadspec` followed by a file name. The full path to the file must be written if it is not located in the directory from which Fastest was started. In our example we have:

```
loadspec bank.tex
```

It is assumed that the file is a text file containing the full specification. If the specification contains syntactic or type errors it will not be loaded and errors will be informed. After loading a specification the user has to select one or more schemas to be tested. Only schemas representing operations can be selected. To select the schema named *Withdraw* just run:

```
selop Withdraw
```

Now we can either queue some testing tactics or apply DNF (see Sect. 2.1.4). If we want to apply a testing tactic to a sub-tree and not to the whole tree, we must apply DNF first and then queue the tactic. We think that in most of the cases the second option is the best. Therefore, we apply DNF with:

```
genalltt
```

Which also builds the corresponding testing tree. If we want to see the testing tree we can issue:

```
showtt
```

in which case we get:

```
Withdraw_VIS
|_____Withdraw_DNF_1
|_____Withdraw_DNF_2
|_____Withdraw_DNF_3
|_____Withdraw_DNF_4
|_____Withdraw_DNF_5
```

If we want to see the contents of a test specification:

```
showsch Withdraw_DNF_1
```

in which case we get:

$R = \{\}, G = \{\}$	$R \neq \{\}, G \neq \{\}, \text{dom } G \subset \text{dom } R$
$R = \{\}, G \neq \{\}$	$R \neq \{\}, G \neq \{\}, (\text{dom } R \cap \text{dom } G) = \{\}$
$R \neq \{\}, G = \{\}$	$R \neq \{\}, G \neq \{\}, \text{dom } R \subset \text{dom } G$
$R \neq \{\}, G \neq \{\},$ $\text{dom } R = \text{dom } G$	$R \neq \{\}, G \neq \{\}, (\text{dom } R \cap \text{dom } G) \neq \{\},$ $\neg (\text{dom } G \subseteq \text{dom } R),$ $\neg (\text{dom } R \subseteq \text{dom } G)$

Figure 6: Standard partition for  $R \oplus G$

```
\begin{schema}{Withdraw\_ DNF\_ 1}\\
  clients : UID \pfun NAME \\
  balances : ACCNUM \pfun BALANCE \\
  owners : UID \rel ACCNUM \\
  u? : UID \\
  n? : ACCNUM \\
  m? : \nat
\where
  u? \mapsto n? \in owners \\
  n? \in \dom balances \\
  m? > 0 \\
  m? \leq balances~n?
\end{schema}
```

Since the operation uses  $\oplus$  which usually is difficult to implement, we will test its implementation by partitioning  $Withdraw_1^{DNF}$  with SP applied to the expression  $balances \oplus \{n? \mapsto balances~n? - m?\}$ . The standard partition delivered with Fastest for  $\oplus$  is depicted in Figure 6. The command is as follows:

```
addtactic Withdraw_DNF_1
  SP
  \oplus
  balances \oplus \{n? \mapsto balances~n? - m?\}
```

In our opinion it does not make too much sense partitioning the remaining test specifications— $Withdraw_2^{DNF}$  to  $Withdraw_5^{DNF}$ —with the same testing tactic because the implementation of  $\oplus$  would not be exercised if their conditions are met. For instance, if  $Withdraw_3^{DNF}$  is true, i.e. if  $m? \leq 0$  holds, it is very unlikely that the program will go through the implementation of  $\oplus$  since inputs are usually checked right at the beginning of the subroutine. In other words, it would be a really awkward program the one in which the balance of the account is updated and then it is checked whether the amount is positive—and in that case the transaction is reversed.

At this point, we can queue another testing tactic or we can run **genalltt** to see the result of applying SP. An expert user would know the results a priori so he/she would add another testing tactic. In this case, we would like to test large withdrawals. Therefore, we can apply NR to  $m?$  to partition all the test specifications resulting from the application of SP. By running the following command:

```
addtactic Withdraw_DNF_1
  NR
  m?
  \langle 100000, 1000000 \rangle
```

```

Withdraw_VIS
|_____Withdraw_DNF_1
|  |_____Withdraw_SP_1
|  |  |_____Withdraw_NR_1
|  |  |_____Withdraw_NR_2
|  |  |_____Withdraw_NR_3
|  |  |_____Withdraw_NR_4
|  |  |_____Withdraw_NR_5
|  |
|  |_____Withdraw_SP_2
|  |  |_____Withdraw_NR_6
|  |  |_____Withdraw_NR_7
|  |  |_____Withdraw_NR_8
|  |  |_____Withdraw_NR_9
|  |  |_____Withdraw_NR_10
.....

```

Figure 7: Part of the testing tree for *Withdraw*

NR will be applied to all the test specifications that will be generated when SP is finally applied—recall that for the moment SP has been queue, not applied. The list of values should have been chosen considering both business and technological issues. If **genalltt** is run both testing tactics are applied in the order they were queued—DNF is applied only the first time this command is issued. The first part of the output of **showtt** is shown in Figure 7. We have omitted the rest because it is too long and it does not add more to the understanding of what is going on. The testing tree has six more branches like  $Withdraw_2^{SP}$  plus  $Withdraw_2^{DNF}$  to  $Withdraw_5^{DNF}$ .

The final step is to run command **genalltca** to generate test cases.

## 4 Tips on Writing Z Models for Fastest

Z is a very general language that can be used for several purposes and specifications can be written in a variety of styles. Although Fastest can work with any kind of Z specifications—provided they are written with the subset of Z currently supported (Appendix A)—it works better if specifications follow some specific rules described in the following sections.

**Important!!** Working differently as we suggest might lead to performance penalties or even to Fastest being unable to derive abstract test cases. If the specification or some of its operations are very complex, then Fastest could crash when these operations have to be processed. However, if complex operations are treated correctly Fastest might provide very useful information regarding test case design and even abstract test cases.

### 4.1 The Z Notation Is Not Fully Supported

Before writing a Z specification for Fastest, please, read Appendix A to learn what parts of the Z notation are still unsupported by the tool. The support for these features is being postponed since we consider that they are somewhat superfluous for Fastest’s purpose. Hence, you will get a broad idea of what kind of models are best suited for Fastest by first reading the appendix.

### 4.2 Fastest Conforms to the Z ISO Standard

Fastest parses and type-checks specifications written in the  $\text{\LaTeX}$  mark-up that conforms to the Z ISO Standard [ISO02]. Then, it does not accept Spivey’s grammar.

### 4.3 Fastest Is Meant to be Used for Unit Testing

The main and foremost purpose of Fastest is to be an aid in **unit testing**. Then, specifications should represent units of implementation or they can be decomposed as such. To be more clear, we think in an unit as a subroutine, a function, or a method. Therefore, if your model ends with schema *System* representing the behaviour of the whole system, and you try to “test” *System*, Fastest will probably perform poorly. Likely, *System* is the disjunction of a number of primitive operations each of which, probably, represents a unit of implementation. Hence, you will get the best of Fastest by trying to “test” each of these operations in isolation instead of working directly with *System*.

One important point here is that software design—i.e. decomposing the software into a set of elements, assigning a function to each element and defining their relationships [GJM03]—should guide the specification. In doing so, you will identify a set of modules or components each providing a public interface to the rest of the system. Each module should be simple and small enough to be easily understood; it will probably be implemented by one programmer. The Z specification of such a design should, then, has one state schema per module and one operation schema for each subroutine exported by each module. These are the primitive operations. If you do not have a design or if you do not want to define one before understanding the requirements, then at least, write the specification as a set of primitive operations that are progressively integrated to provide some complex services. In either case, use Fastest to derive test cases for these primitive operations.

Note that full specifications can be given for these primitive operations. In other words, give schemas for both successful and erroneous conditions for each operation, but keep them primitive.



$$A_1 == B \wedge C$$

$$A_2 == B \circ C$$

$$A_3 == (B \vee D) \wedge C$$

$$\begin{array}{|l} \hline A_4 \\ \hline Decl \\ \hline Pred \Rightarrow B' \\ Pred_1 \Rightarrow C' \\ \hline \end{array}$$

$$A_5 == [Decl \mid Pred] \wedge B \wedge C$$

$$\begin{array}{|l} \hline A_6 \\ B \\ C \\ Decl \\ \hline Pred \\ \hline \end{array}$$

Figure 8: Some compound operations. *Decl* is a declaration and *Pred* and *Pred<sub>1</sub>* are a predicates. *B*, *C* and *D* are primitive operations.

#### 4.4 Be Careful in Testing Compound Operations

Although it is a matter of style, specifiers might want to represent that some operation “calls” or “uses” the services of other operations. This is some times achieved by specifying an operation as the conjunction ( $\wedge$ ) or the composition ( $\circ$ ) of some other operations. In particular, conjunction can be written as schema inclusion. We call these *compound operations*. For instance, consider the compound operations sketched in Figure 8. Let’s assume in all cases that *B*, *C* and *D* are primitive operations. Besides, say that *fA1*, *fA2*, *fA3*, *fA4*, *fA5*, *fA6*, *fB*, *fC* and *fD* are the subroutines implementing the corresponding operations. Then, we suggest to work as follows in each case.

**Cases *A<sub>1</sub>* and *A<sub>2</sub>*.** These cases are very easy to deal with. Just derive test cases for *B* and *C* and not for *A<sub>1</sub>* and *A<sub>2</sub>*. The point here, as with some of the other cases, is that the correctness of either *fA1* or *fA2* depends solely on the correctness of both *fB* and *fC*. Then, one should derive unit test cases for *fB* and *fC* only and then integrate<sup>2</sup> them to test *fA1* and *fA2*. Since Fastest now is only good for unit testing, then trying to apply it to derive test cases for *A<sub>1</sub>* and *A<sub>2</sub>* might not be the best option.

**Case *A<sub>3</sub>*.** We have distinguished this case from the previous one because we want to emphasize that we think that *B* and *D* are two distinct operations and not two schemas of the same operation—like the normal case and an erroneous one. If this is the case, then we think that the best course of action is to work as indicated in the previous paragraph.

**Case *A<sub>4</sub>*.** *B'* and *C'* do not have precondition since all of their variables are primed. Hence, their predicates will not have much influence in abstract test case derivation since test cases are generated from the input space of the operation. For this reason Fastest will not unfold these schema references. In this case, then, the user can work directly with *A<sub>4</sub>*.

**Case *A<sub>5</sub>*.** We suggest to write *A<sub>5</sub>* as follows:

$$A_5 == E \wedge B \wedge C$$

where *E* is  $[Decl \mid Pred]$ , and then to derive test cases for *B*, *C* and *E* and not for *A<sub>5</sub>*. If  $[Decl \mid Pred]$  is not named, Fastest will not recognize it as an operation thus making it impossible for the user to work with it.

<sup>2</sup>Integration testing is not implemented yet.

$x$  is intended to be a natural number but  $\mathbb{N}$  is not a type, then we need an invariant. An equivalent schema would have been  $Naturals == [x : \mathbb{Z}]$  but the invariant would be hidden.

$Naturals$
$x : \mathbb{Z}$
$0 \leq x$

$Decr$
$\Delta Naturals$
$x' = x - 1$

No proof is needed to verify that  $Decr$  preserves the state invariant because state variables are restricted to satisfy it.

(a) Classic style.

$Naturals$
$x : \mathbb{Z}$

$NaturalsInv$
$Naturals$
$0 \leq x$

$Decr$
$\Delta Naturals$
$x > 0$
$x' = x - 1$

**Theorem**  $DecrVerifiesInvariant$

$$NaturalsInv \wedge Decr \Rightarrow NaturalsInv'$$

(b) Proof obligation style.

Figure 9: A simple example showing two styles of writing state invariants. Fastest works better with the proof obligation style.

**Case  $A_6$ .** As with the previous case we suggest to rewrite  $A_6$  as follows:

$$E == [Decl_1 \mid Pred]$$

$$A_6 == E \wedge B \wedge C$$

where  $Decl_1$  might be different from  $Decl$  since it may be necessary to add some variables because  $E$  does not include  $B$  and  $C$ , which may add some declarations. If the operation is written in this way, then derive test cases for  $B$ ,  $C$  and  $E$  and not for  $A_6$  as we suggested in the previous cases.

## 4.5 Do Not Include the State Invariant in the State Schema

It is the classic style within the Z community to include the state invariant inside the state schema, as shows the simple example of Figure 9a. However, Fastest works better if the state invariant is not included in the state schema as shown in Figure 9b. This is the style followed by specification languages such as B [Abr96] and TLA+ [Lam02].

Writing the state invariant outside the state schema makes it a proof obligation rather than a state restriction. At the same time, this style avoids implicit preconditions perhaps making the specification clearer to programmers because they do not need to calculate them. But explicit preconditions are the key to input domain partition, which is the fundamental concept behind the TTF. Hence, by writing the state invariant outside the state schema we avoid implicit preconditions, thus, enabling input domain partition.

## 4.6 Keep the State Schema Focused on a Set of Related Operations

As we have seen, the input space (IS) of a Z operation is defined as the schema declaring all the input and state variables of the operation. An abstract test case is an element belonging to the

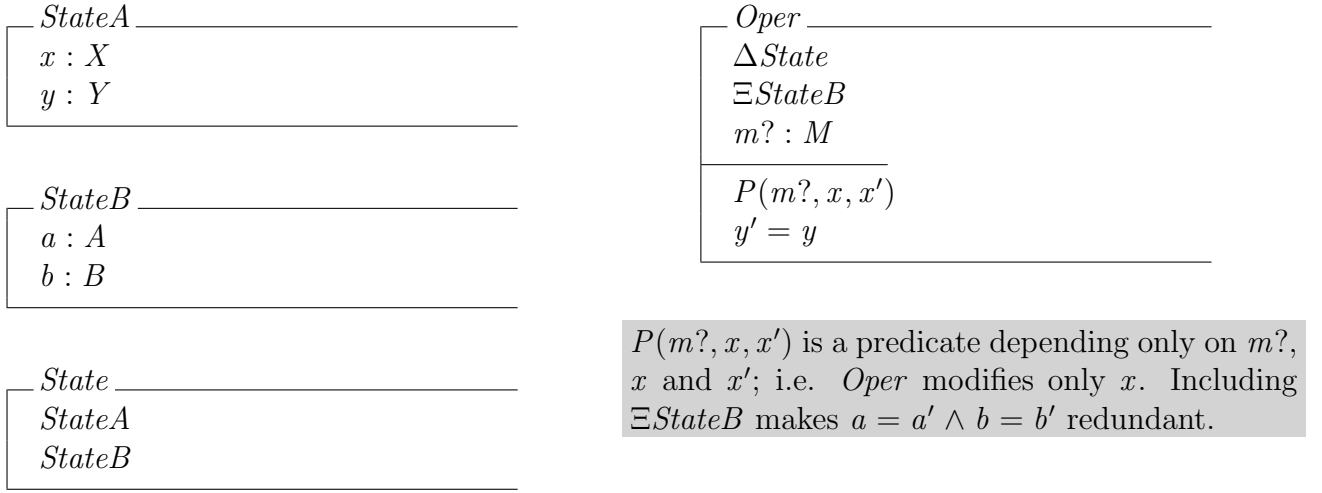


Figure 10: State variables are grouped in state schemas. Operations include  $\Delta$  of the full state and  $\Xi$  of those part of the state that they do not modify.

IS. Then, the more variables in the IS, the longer the abstract test cases. Furthermore, if some of the IS variables are not referenced in the operation's predicate, then it means that these variables are irrelevant for the operation. But they still need to be included in abstract test cases. Hence, it is possible that abstract test cases contain many variables such that only a fraction of them are meaningful to the tester.

It does not matter whether this irrelevant variables appear in predicates such as  $var' = var$ , there still be an equality like  $var = const$  in every abstract test case of the corresponding operation. For instance, it is a common style to divide the state variables in some state schemas that then are joined to define the whole state space of the system, as shown in 10. In this way, specifiers avoid to write many equalities for those variables that the operation does not modify. However, a better strategy if the specification is going to be loaded into Fastest would be to specify  $\text{Oper}$  as follows:

$$\text{Oper} == [\Delta \text{StateA}; m? : M \mid P(m?, x, x') \wedge y' = y]$$

Clearly, this is possible only because  $\text{Oper}$ 's predicate depends only on the state variables declared in  $\text{StateA}$ . In the extreme case  $\text{Oper}$  can be specified with:

$$\text{Oper} == [x, x' : X; m? : M \mid P(m?, x, x')]$$

but this implies that abstract test cases derived by Fastest will not mention  $y$ . This might be a problem if the unit implementing  $\text{Oper}$  needs some initial value for all of its variables—but perhaps that is an indication of some poor implementation.

## 4.7 Avoid Using Quantifiers

Quantifiers always complicate software verification. Then, it is a good advice to avoid them as much as possible regardless whether Fastest will be used or not—wisely use the rich mathematical operators provided by Z to avoid many quantifications. Fastest will enter into troubles if it needs to find abstract test cases from test classes whose predicates include quantifiers. However, it will succeed in many cases.

## 4.8 Avoid Axiomatic Definitions

Fastest supports axiomatic definitions as described in Section 5.4. However, their presence decreases the level of automation of the tool, so it is better to avoid them as much as possible. Conceptually,

axiomatic definitions are parameters of the specification. In other words, the meaning of an specification depends on the particular value assumed for each axiomatic definition. Since test cases are derived from the specification, they are also parametrized by its axiomatic definitions. Therefore, the user first needs to set a constant value for each axiomatic definition and then Fastest derives test cases considering those values. In this way, the application is tested for only one of its possible meanings.

## 4.9 Avoid Arbitrary Numeric Constants

If *memSize* stands for the amount of available memory of some computer and the specification includes an axiomatic definition such as:

$$\frac{memSize : \mathbb{N}}{memSize = 1024}$$

and an operation like:

$$\frac{\begin{array}{l} WriteMemOk \\ \Delta MemoryState \\ x? : BYTE \end{array}}{\begin{array}{l} \#mem < memSize \\ mem' = mem \frown \langle x? \rangle \end{array}}$$

then possible test classes cases may be:

- $mem = \langle \rangle$
- $\#mem = 1$
- $\#mem = memSize - 1$
- $\#mem = memSize$
- $\#mem = memSize + 1$

and possible corresponding abstract test cases may be:

- $x? = byte0 \wedge mem = \langle \rangle$
- $x? = byte0 \wedge mem = \langle byte0 \rangle$
- $x? = byte0 \wedge mem = \langle byte0, \dots, byte0 \rangle$  where ... represents 1021 elements
- $x? = byte0 \wedge mem = \langle byte0, \dots, byte0 \rangle$  where ... represents 1022 elements
- $x? = byte0 \wedge mem = \langle byte0, \dots, byte0 \rangle$  where ... represents 1023 elements

It is very important to remark that, although we have used “...” to represent elements in each sequence, in the real abstract test cases the elements must be written down. Precisely, the difficulty in writing down those test cases is the reason for which we suggest avoiding arbitrary numeric constants. Fastest will automatically find abstract test cases for all the above test classes, but there are many situations where it will fail.

If the model is slightly rewritten Fastest will have more chances to automatically derive abstract test cases for more test classes and, likely, the implementation will be verified as thoroughly as with the first model. The only change is to avoid the constant by rewriting the axiomatic definition as follows:

$$\left| \begin{array}{l} memSize : \mathbb{N} \\ \hline 0 < memSize \end{array} \right|$$

Then, users can derive the same test classes and later they bind a smaller constant to *memSize*, for example 10.

If the implementation can be *configured* to assume that the available memory is 10 bytes, then it is very likely that these test cases will uncover the same errors than the original ones—and perhaps in less time. It is important to remark that this will work if the implementation is configured, and not modified. In other words, it will work if there is some symbolic constant that can be modified without changing a single line of code or if this value is returned by some external function.

## 4.10 Do Not Use Total Functions Over Infinite Sets

A total function whose domain is an infinite set is an infinite set. Fastest cannot generate test cases involving infinite sets. Hence, no specification including a total function over an infinite set should be provided to Fastest if it is expected that it will generate test cases for that function. Replace it with a partial function or by a total function whose domain is finite (and not very large).

## 5 User's Manual

Throughout of this manual remember that Fastest is still a prototype. Then, it is not as robust as it should be.

For an academic presentation of Fastest see [CAF<sup>+</sup>14].

### 5.1 Installing and Executing Fastest

Fastest should work on any environment with Java SE Runtime Environment 1.6 or newer. However, it has been tested only on Linux and MS-Windows boxes. To install the tool, just decompress and unarchive the file `Fastest.tar.gz` in any folder of your choice.

#### 5.1.1 Running Fastest and Entering Commands

To run Fastest in application mode, open a command window and run one of the following commands, where `INSTALLDIR` is the full path to the directory where Fastest was installed.

```
java -jar INSTALLDIR/fastest.jar
```

```
java -Xss128k -Xms512m -Xmx512m -jar INSTALLDIR/fastest.jar
```

The first command will serve for most purposes, but if large specifications will be used then the second command is a better option. If the computer has at least 1 Gb of memory then the second option should be used. The `Xms` and `Xmx` options indicate the minimum and maximum amounts of memory that the Java process will be able to use, respectively. Then, if more memory is needed increase the maximum (it must be a multiple of 1024). In this version of Fastest it is difficult to know if more memory is needed, but one symptom is command `genalltt` (section ??) taking too long—more than one minute—to finish.

In either case, Fastest prints the following prompt from which users can issue commands.

```
Fastest>
```

To enter a command just type-in it along with its arguments and then press the return key. There are three ways of learning which commands are available:

- Type `help` and press the return key.
- Press the TAB key and a list of commands will be printed.
- Type-in the first letters of a command and then press the TAB key: either a list of the commands whose name starts with those letters will be printed, or the complete command name, if any, will be printed. For instance, if after entering the `a` key the TAB key is pressed the following is printed (`[TAB]` means pressing the TAB key):

```
Fastest>a[TAB]
```

```
addtactic apply
```

```
Fastest>a
```

And if the `d` key is pressed followed by the TAB key again, the result is the whole `addtactic` command printed, as follows:

```
Fastest>ad[TAB]dtactic
```

When the TAB key is pressed after command `loadspect`, Fastest prints the contents of the working directory. For example, if Fastest is run from the installation directory, the result is as follows:

```
Fastest>loadspect [TAB]
```

```
doc                fastest-server.jar    fastest.jar
lib
Fastest>showsched
```

If the user types-in the first letters of one these files or directories and then presses the TAB key again, the name will be completed or a filtered list will be displayed, as with command names. If the letters correspond to a file name and it is completed, a blank space is added at the end; but if the letters correspond to a directory, when the name is completed a / or \ character is added at the end. If the user presses the TAB key again, the content of this directory is displayed. The TAB key can be further pressed as a means of exploring the contents of the inner directories.

The left and right arrow keys can be used to move the cursor along the line being edited to modify it by inserting or deleting any character. The up and down arrow keys move across the commands that have been issued during the session. If one of these commands is recovered the user can modify it by using the left and right arrow keys, and can run it again by pressing the return key. Commands are executed when the return key is pressed regardless of where the cursor is.

**Important!!** Note that `Ctrl+C` kills the program making all the data and commands to be lost. Future versions will be more robust.

**Important!!** Fastest does not save anything by default. The user has to use one of the commands described in section 5.9 to save the data generated during a session.

## 5.2 Steps of a Testing Campaign

Roughly speaking, currently, a typical testing campaign carried on with Fastest can be decomposed in the steps listed below. Some of them are optional and some can be executed in a different order, as is described in the referred sections between brackets. Also, at any time users can run commands to explore the specification and the testing trees, and to save their results (5.9).

1. Load the specification (5.3).
2. Select the operations to be tested (5.3).
3. Set a value for each axiomatic definition (5.4.6).
4. Select a list of testing tactics to be applied to each operation (5.5).
5. Generate testing trees (one for each selected operation) (5.5).
6. Calculate abstract test cases and prune unsatisfiable test classes (5.7).
7. If some leaves do not have an abstract test case, then explore these leaves to determine the cause for that. There are two possible causes:
  - The leaf predicate is a contradiction, but Fastest failed in pruning it.

- The leaf predicate is not a contradiction, but Fastest was not smart enough to find an abstract test case for it. In this case you can try to reduce the size of some sets or integer constants.

8. If all of the leaves have an abstract test case, then save the results (5.9) and leave the program (5.10).

Step 4 is perhaps the most relevant step of all since it will determine how revealing and leafy testing trees are going to be.

Steps 4 and 5 can be executed iteratively and in the specified order or the opposite one.

Test case design includes from step 1 to step 5. The remaining steps generate test data, i.e. abstract test cases.

The following sections explain in detail each of the steps of a testing campaign carried on with Fastest.

### 5.3 Loading a Specification and Selecting Schemas

An specification is loaded by executing `loadspec` followed by a file name. The full path to the file must be written if it is not located in the directory from which Fastest was started, as in the following example:

```
Fastest> loadspec doc/sensors-simp.tex
```

It is assumed that the file is a text file containing the full specification; the current version does not support the  $\text{\LaTeX}$  directive `\input{}`. If the specification contains syntactic or type errors it will not be loaded and the errors will be informed. It is possible to load only one specification at a time. To load a new specification run the `loadspec` command again or reset the current session by running command `reset` (in either case all the data generated so far will be lost).

It is possible to load specifications where terms are used before their declarations.

Once a specification has been loaded, it can be explored, printed and saved with the commands described in section 5.9.

After loading a specification the user has to select one or more schemas to be tested. Only schemas representing operations can be selected. A schema represents an operation if it contains any combination of the following: (a) an input or unprimed state variable, or (b) an output or primed state variable. To select an schema use `selop` followed by the name of a Z schema representing an operation. The list of candidate Z schemas can be displayed with `showloadedops`, with no arguments. It can be selected as many schemas as needed by issuing the same number of `selop` commands. A schema that was previously selected can be deselected with command `deselop` followed by its name.

### 5.4 Dealing with Axiomatic Definitions

According to the TTF and to the semantics of the Z notation, identifiers declared in axiomatic definitions neither are state variables nor input variables. However, since they do appear in operations, they are carried all the way down to test classes. Hence, when abstract test cases have to be derived from test classes it is necessary to bind a value for each identifier declared in an axiomatic definition, because otherwise there is no way to find a tuple of values satisfying the test class' predicate—in this sense, Fastest treats all these identifiers as model parameters. At the same time, some axiomatic definitions may be complex predicates that must be considered when test classes and cases are generated. In summary, Fastest requires that all the axiomatic definitions appearing in at least one of the selected operations must be replaced by a constant value or by its definition, before testing tactics are applied. In this way, testing trees will not include axiomatic definitions.



$[CHAR, USER]$

$control, blank : \mathbb{P} CHAR$ $null, tab, space : CHAR$ $Max, Mid, Min : \mathbb{Z}$ $root : USER; adm, audit : \mathbb{P} USER$ $asciiTbl : \mathbb{N} \rightarrow CHAR$
$null \in control$ $control \cap blank \neq \emptyset$ $Min = 34$ $Max = 1000 + Min$ $blank = \{tab, space\}$ $root \in adm$ $adm = audit \cup \{root\}$

Figure 11: Examples of identifiers declared in an axiomatic definition.

Fastest replaces axiomatic definitions by their values or definitions with command `replaceaxdef` but, for some axiomatic definitions, users need to run command `setaxdef` before running `replaceaxdef`. Users can see the identifiers for which Fastest automatically bound a constant value by running command `showaxdefvalues`. So, we recommend to first read the following sections.

Fastest classifies identifiers declared in axiomatic definitions in the following categories, treating each of them in a different way as is described below.

#### 5.4.1 Basic Types

An identifier,  $ident : T$  where  $T$  is a basic type, declared in an axiomatic definition is considered a constant. For example  $null, tab, space$  and  $root$  in Figure 11 are considered to be constants of their respective types. These constants are used when Fastest calculates abstract test cases (5.7). The user does not need to take any action for these identifiers.

#### 5.4.2 Symbolic Constants

A identifier,  $ident : T$  where  $T$  can be any type but a basic one, declared in an axiomatic definition is a *symbolic constant* if there is exactly one equality of the form  $ident = cexpr$ , where  $cexpr$  is a *constant expression*. A constant expression is any valid expression verifying any of the following:

- The expression is a number or an element of an enumerated type.
- The expression includes only symbolic constants, numbers, elements of enumerated or basic types and  $\mathbb{Z}$  symbols.

For example, in Figure 11,  $Min, Max$  and  $blank$  are symbolic constants.  $Mid$  is not a symbolic constant because there is no equality defining a constant value for it; and  $adm$  is not a symbolic constant neither because  $audit \cup \{root\}$  is not a constant expression.

When `replaceaxdef` is run Fastest automatically replaces all the symbolic constants by their corresponding values. In particular, for example,  $Min$  is replaced by 34 in  $1000 + Min$  so  $Max$  is replaced by  $1000 + 34$  when it is replaced in a schema. Then, users do not need to run `setaxdef` for this kind of axiomatic definition.

### 5.4.3 Equalities

If an identifier,  $ident : T$  where  $T$  is any type, is declared in an axiomatic definition, and there is exactly one equality of the form  $ident = expr$ , where  $expr$  is not a constant expression, then users should bind a value for each identifier in  $expr$  for which Fastest does not automatically bind a constant value and, at the same time, they cannot bind a value to  $ident$ . Then, when `replaceaxdef` is run, Fastest automatically replaces  $ident$  by  $expr$  (which now is a constant expression) in all the schemas. If some variable in  $expr$  remains unset, Fastest will return an error message when `genalltca` is run and all testing trees will need to be recreated.

For example, users need to manually bind a value to *audit* in Figure 11 but they cannot bind a value to *adm*.

Users can bind values to identifiers with command `setaxdef` which is explained in Sect. 5.4.6.

### 5.4.4 Equivalences

An equivalence is any axiomatic definition matching the following:

$$\frac{x : T}{\forall y : U \bullet P(x, y) \Leftrightarrow Q(y)}$$

where  $T$  and  $U$  are any types,  $P$  is an atomic predicate and  $Q$  is any predicate.  $Q$  may depend also on other axiomatic descriptions. We say  $Q$  is the definition of  $x$ . The following definition falls in this category:

$$\frac{failed : \mathbb{P}((TIME \rightarrow PVAL) \times PVAL \times CheckDef)}{\forall h : TIME \rightarrow PVAL; v : PVAL; d : CheckDef \bullet (h, v, d) \in failed \Leftrightarrow avrDelta(lastRepVal h v d.rep) < d.low}$$

`replaceaxdef` replaces all the uses of axiomatic definitions of this kind by their corresponding predicates. In doing so, formal parameters (for instance  $h$ ,  $v$  and  $d$  in *failed*) are replaced by the actual parameters appearing in the use being replaced. If an axiomatic definition in this category depends on other axiomatic definitions, then these must have a constant value before `replaceaxdef` is run. In this regard, users must do as explained in previous sections.

### 5.4.5 All Other Declarations

Identifiers declared in axiomatic definitions that do not meet the conditions described in the previous sections fall in this category<sup>3</sup>. For these identifiers the user should give constant values so Fastest has chances to find abstract test cases for all the test classes.

The user can bind values to identifiers with command `setaxdef` (5.4.6).

### 5.4.6 Command setaxdef

Fastest provides command `setaxdef` to bind values to identifiers declared in axiomatic definitions—provided they can be bind at all. Command syntax is as follows:

```
setaxdef ident ["constant_declarations"] "value"
```

---

<sup>3</sup>We will further subdivide this category to solve some issues automatically in future releases.

where, **ident** is the identifier for which the user wants to set a constant value and **value** is that value. This means that Fastest will replace the identifier for the value when replacing other axiomatic definitions. The optional parameter **constant\_declarations** must be used when the **value** refers to constants of basic types (see an example below). For example, the following command sets a value for *Mid* (declared in Figure 11):

```
setaxdef Mid "517"
```

When such a command is issued, Fastest checks that the type of the value is consistent with the type of the identifier. Also, it *tries* to check that the value satisfies all of the predicates, appearing in axiomatic definitions, where the identifier is referenced. However, this check can only be finished when all these predicates become constant, i.e. when all the variables have been bound to a constant value. Then, when the last identifier is bound to a constant, the predicate is evaluated and, possibly, an error message is printed. Therefore, if Fastest complains that the value that was last bound to an identifier does not verifies a predicate in an axiomatic definition, the user should check whether this last value is the cause of the problem or it is the values previously bound to the other identifiers. If this is the case, the user can reset the previous values with the same command, until no error messages are printed.

Now, let's see an example involving the optional parameter **constant\_declarations**. Say *asciiTbl* (defined in figure 11) is used in some operation. Then, Fastest needs that the user sets a value for it so the tool can find abstract test cases for all the test classes generated for the operation. In the same axiomatic definition have been defined some *CHAR*'s but, say, the user wants to test the operation with a more realistic ASCII table. Hence, the user can issue a command like this one:

```
setaxdef ident "char0,char1,char2:CHAR"
              "\{0 \mapsto null, 1 \mapsto char0, 2 \mapsto char1,
              3 \mapsto char2, 4 \mapsto tab, 5 \mapsto space\}"
```

In other words, **constant\_declarations** allows the user to declare some constants of basic types that are used to define the constant value to be bound to the identifier. Internally, Fastest declare these identifiers in axiomatic definitions. Although the user can chose any names in the declaration, there are two things worth to mention:

1. Avoid name clashes with other identifiers declared in axiomatic definitions and operations.
2. Chose names that increase the likeness of Fastest finding abstract test cases by following the rules described in section ??.

If constants of different types need to be declared, the syntax is the same than in Z, i.e.:

```
setaxdef ident "char0,char1,char2:CHAR; user0,user7:USER" ...
```

The user can see the values bound to identifiers by running command **showaxdefvalues**. Besides, Fastest provides command **showaxdefs** so users can easily see all the axiomatic definitions used in the specification.

**setaxdef** can be executed right after **loadspect**.

## 5.5 Applying Testing Tactics and Generating Testing Trees

Testing tactics can be applied to any sub-tree of any (previously) selected schema—in particular they can be applied to the entire tree. To apply a testing tactic to a particular sub-tree, that sub-tree must already exist. Then, the first tactic can only be applied to the VIS of the operation. The first tactic applied by Fastest is always Disjunctive Normal Form (DNF, see below). To apply DNF to all the selected schemas just run `genalltt`.

Except for DNF, tactic application is performed in two steps:

1. Add the tactic to the list of tactics to be applied.
2. Run `genalltt`.

These steps can be repeated as many times as needed. It is also possible to run these steps even before `genalltt` is run to apply DNF, in which case Fastest first applies DNF and then the tactics added by the user—DNF is applied only once the first time `genalltt` is run. Testing trees can be displayed with `showtt` (5.9).

**Important!!** If `genalltt` takes more than a couple of minutes to finish it might be the case that the Java process run out of memory. It usually happens when the DNF of an operation has thousands of disjuncts—this, in turn, occurs when the operation is too complex considering full schema unfolding. If this occurs, the program will look like tilt—we hope to solve this in future versions. The only thing the user can do is to kill the process from the operating system. This problem might be solved by augmenting the memory available for the Java process (5.1).

The command `addtactic` adds a testing tactic to the list of tactics to be applied to a particular (previously selected) operation. Tactics are applied in the order they are entered by the user. Initially, the list of tactics of any operation includes only DNF, which is the first to be applied. The command syntax is rather complex because it depends on the tactic that is going to be applied (see the following sections for more details). The base syntax is:

```
addtactic sub_tree tactic_name parameters
```

where `sub_tree` is the name of either a selected schema or the name of a test class already generated, `tactic_name` is the name of a tactic supported by Fastest, and `parameters` is a list of parameters that depends on the tactic.

If `sub_tree` is the name of a schema, the tactic is applied to all the existing leaves of the corresponding testing tree. If `sub_tree` is the name of an existing test class, the tactic is applied to all the leaves of the sub-tree whose root node is that test class. The examples shown in Figure 12 may clarify this behaviour.

Unless `addtactic` prints an error message, the tactic has been successfully added. This command produces no other effect than adding the tactic to an internal list until command `genalltt` is executed.

Command `showtactics` prints a brief description of the available tactics; the following sections describe them in more detail.

### 5.5.1 Disjunctive Normal Form

This tactic is applied by default and it must not be selected with `addtactic`. By applying this tactic the operation is written in Disjunctive Normal Form and the *VIS* is divided in as many test classes as terms are in the resulting operation's predicate. The characteristic predicate of each class is the precondition of one of the terms in the operation's predicate.

```

KeepMaxReading_VIS
!_____KeepMaxReading_DNF_1
!_____KeepMaxReading_DNF_2
!_____KeepMaxReading_DNF_3

genalltt

```

(a) Applying just DNF.

```

KeepMaxReading_VIS
!_____KeepMaxReading_DNF_1
!_____KeepMaxReading_SP_1
!_____KeepMaxReading_SP_2
!_____KeepMaxReading_SP_3
!_____KeepMaxReading_SP_4
!_____KeepMaxReading_SP_5
!_____KeepMaxReading_DNF_2
!_____KeepMaxReading_DNF_3

genalltt
addtactic KeepMaxReading_DNF_1 SP < smax~s? < r?
genalltt

```

(b) Applying DNF and then SP to just one test class.

```

KeepMaxReading_VIS
!_____KeepMaxReading_DNF_1
!_____KeepMaxReading_SP_1
!_____KeepMaxReading_SP_2
!_____KeepMaxReading_SP_3
!_____KeepMaxReading_SP_4
!_____KeepMaxReading_SP_5
!_____KeepMaxReading_DNF_2
!_____KeepMaxReading_SP_6
!_____KeepMaxReading_SP_7
!_____KeepMaxReading_SP_8
!_____KeepMaxReading_SP_9
!_____KeepMaxReading_SP_10
!_____KeepMaxReading_DNF_3
!_____KeepMaxReading_SP_11
!_____KeepMaxReading_SP_12
!_____KeepMaxReading_SP_13
!_____KeepMaxReading_SP_14
!_____KeepMaxReading_SP_15

addtactic KeepMaxReading SP < smax~s? < r?
genalltt

```

(c) Applying DNF and SP to the entire testing tree.

```

KeepMaxReading_VIS
!_____KeepMaxReading_DNF_1
!_____KeepMaxReading_SP_1
!_____KeepMaxReading_SP_2
!_____KeepMaxReading_SP_3
!_____KeepMaxReading_SP_4
!_____KeepMaxReading_SP_5
!_____KeepMaxReading_DNF_2
!_____KeepMaxReading_DNF_3
!_____KeepMaxReading_NR_1
!_____KeepMaxReading_NR_2
!_____KeepMaxReading_NR_3
!_____KeepMaxReading_NR_4
!_____KeepMaxReading_NR_5

genalltt
addtactic KeepMaxReading_DNF_1 SP < smax~s? < r?
addtactic KeepMaxReading_DNF_3 NR r? \langle 10, 1000 \rangle
genalltt

```

(d) Applying DNF and then two different tactics to two different test classes.

Figure 12: In each figure we show the testing tree produced with the script shown below them (scripts include only the relevant commands).

### 5.5.2 Standard Partition (Fastest's name SP)

This tactic uses a predefined partition of some mathematical operator (see “Standard domains for Z operators” at page 165 of Stocks’ PhD thesis [Sto93]).

Take a look at Appendix C and at the file `INSTALLDIR/lib/conf/stdpartition.spf` to see what standard partitions are delivered with Fastest and how to define new ones. We think the syntax is rather straightforward. The user can edit this file to change, erase or add standard partitions, thus making this tactic quite powerful and flexible. Fastest needs to be restarted if this file is changed because it is loaded only during start up.

To apply one of those standard partitions to an operation the command is as follows.

```
addtactic op_name SP operator expression
```

where **operator** is the  $\text{\LaTeX}$  string of a Z operator and **expression** is a Z expression written in  $\text{\LaTeX}$ . It is assumed that **operator** appears in the **expression** and this in turn appears in the predicate of the selected operation. Hence, this tactic can be applied to different operators and different expressions of the same operation.

The application of the tactic divides each test class at a given level of the testing tree in as many test classes as conjunctions defines the partition. Each conjunction is conjoined to the predicate of the test class being partitioned to form a new test class.

### 5.5.3 Free Type (Fastest's name FT)

This tactic generates as many test classes as elements a free type (enumerated) has. In other words if a model defines type  $COLOUR ::= red \mid blue \mid green$  and some operation uses  $c$  of type  $COLOUR$ , then by applying this tactic each test class will be divided into three new test classes: one in which  $c$  equals  $red$ , the other in which  $c$  equals  $blue$ , and the third where  $c$  equals  $green$ .

The tactic is applied with the following command:

```
addtactic op_name FT variable
```

where **variable** is the name of a variable whose type is a free type.

Currently, Free Type works only if the free type is actually an *enumerated* type, i.e. an inductive type defined only by constants.

### 5.5.4 In Set Extension (Fastest's name ISE)

It applies to operations including predicates of the form  $expr \in \{expr_1, \dots, expr_n\}$ . In this case, it generates  $n$  test classes such that  $expr = expr_i$ , for  $i$  in  $1 \dots n$ , are their characteristic predicates. The command to add this tactic is as follows:

```
addtactic op_name ISE predicate
```

where **predicate** is an atomic predicate of the form shown above.

### 5.5.5 Proper Subset of Set Extension (Fastest's name PSSE)

This tactic uses the same concept of ISE but applied to set inclusions. PSSE helps to test operations including predicates like  $expr \subset \{expr_1, \dots, expr_n\}$ . When PSSE is applied it generates  $2^n - 1$  test classes whose characteristic predicates are  $expr = A_i$  with  $i \in 1 \dots 2^n - 1$  and  $A_i \in \mathbb{P}\{expr_1, \dots, expr_n\} \setminus \{\{expr_1, \dots, expr_n\}\}$ .  $\{expr_1, \dots, expr_n\}$  is excluded from  $\mathbb{P}\{expr_1, \dots, expr_n\}$  because  $expr$  is a proper subset of  $\{expr_1, \dots, expr_n\}$ . The command syntax is as follows:

```
addtactic op_name PSSE predicate
```

where **predicate** is an atomic predicate of the form shown above.

### 5.5.6 Subset of Set Extension (Fastest's name SSE)

It is similar to PSSE but it applies to predicates of the form  $expr \subseteq \{expr_1, \dots, expr_n\}$  in which case it generates  $2^n$  by considering also  $\{expr_1, \dots, expr_n\}$ . The command syntax is as follows:

```
addtactic op_name SSE predicate
```

where `predicate` is an atomic predicate of the form shown above..

### 5.5.7 Numeric Ranges (Fastest's name NR)

With this tactic the user can bind an ordered list of numbers,  $n_1, \dots, n_k$ , to a numeric variable,  $var$ , in such a way that, when the tactic is applied, it generates  $2 * k + 1$  test classes characterized by the following predicates:  $var < n_1$ ,  $var = n_1$ ,  $n_1 < var < n_2$ ,  $\dots$ ,  $var = n_i$ ,  $n_i < var < n_{i+1}$ ,  $var = n_{i+1}$ ,  $\dots$ ,  $var < n_k$ ,  $var = n_k$  and  $n_k < var$ . Consider the following example.

Variable appearing in operation	$memPointer : \mathbb{N}$
List provided by the user	$\langle 0, 65535 \rangle$
Test classes generated by the tactic	$T_1 \rightarrow memPointer < 0$
	$T_2 \rightarrow memPointer = 0$
	$T_3 \rightarrow 0 < memPointer \wedge memPointer < 65535$
	$T_4 \rightarrow memPointer = 65535$
	$T_5 \rightarrow 65535 < memPointer$

The command to apply this tactic is as follows:

```
addtactic op_name NR variable \langle list of numbers \rangle
```

where `variable` is the name of a numeric variable appearing in the operation; and each element in the list must be separated by a comma and in increasing order. The list must be non empty. If the type of the variable is  $\mathbb{N}$ , Fastest checks that all the numbers in the list are naturals.

**Important!!** In this version, Fastest will accept lists of numbers in any order but the behaviour of the tactic will be unpredictable.

### 5.5.8 Mandatory Test Set (Fastest's name MTS)

With this tactic the user can bind a set of constants,  $\{v_1, \dots, v_n\}$  to an expression,  $expr$ , in such a way that, when the tactic is applied, it generates  $n + 1$  test classes characterized by the following predicates:  $expr = v_i$  for all  $i$  in  $1 \dots n$ , and  $expr \notin \{v_1, \dots, v_n\}$ .

The command to apply this tactic is as follows:

```
addtactic op_name MTS "expr" set_extension
```

where `expr` is an expression appearing in the operation and `set_extension` is a set extension, written in  $\text{\LaTeX}$  mark-up, whose members are constants. Fastest checks whether the types of `expr` and `set_extension` are consistent.

In this version, the constants in `set_extension` can be numbers, elements of enumerated types, identifiers declared in axiomatic definitions or constants assembled out of them—for instance,  $2 \mapsto ON$ , where  $ON$  is an element of an enumerated type.

## 5.6 Manually pruning the testing tree

Test classes can be pruned not only because they are empty, but also because they will not give meaningful test cases. This is at the engineer discretion. Fastest provides commands `prunefrom` and `prunebelow` to erase a sub-tree from some testing tree; and command `unprune` to restore previously erased sub-trees. Their syntax and semantics are as follows.

- `prunefrom class_name`

This command deletes the sub-tree hanging from and including `class_name`. It is useful to erase leaves.

- `prunebelow class_name`

This command deletes the sub-tree hanging from but not including `class_name`.

- `unprune class_name`

This command restores the sub-tree hanging from and including `class_name`. Note that it is impossible to restore a sub-tree hanging from a pruned test class.

It is important to remark that manually pruning test classes from a testing tree can reduce the quality of testing. This happens when the tester prunes one or more classes that are not empty, and thus Fastest will not generate abstract test cases for them.

## 5.7 Generating Abstract Test Cases

`genalltca` finds abstract test cases for the leaves of the testing trees and prunes unsatisfiable test classes.

**Important!!** With the default configuration values, Fastest will try to find a test case for each test class for at most 10 seconds. Then, if there are many leaves this process can take a long time to finish. The tool will remain useless until the whole process terminates. If the process is interrupted, it will have to be restarted from the very beginning. In that case all the abstract test cases generated so far will be lost.

Besides generating abstract test cases, the output of this command is a series of messages printed on the screen. For each test class being analysed, the command first prints a message like this one:

Trying to generate a test cases for the class: `<tcn>`

where `tcn` is the name of the test class. After some time Fastest will print one of the following messages for each test class:

- If Fastest found an abstract test case for a given test class, the following message is written:

```
<tcn> test case generation -> SUCCESS.
```

- If Fastest was unable to find an abstract test case it will print the following message:

```
<tcn> test case generation -> FAILED.
```

Once `genalltca` finishes, the user can explore and save the abstract test cases with command `showsch -tca` (5.9). Then, the user has to analyse those test classes for which a **FAILED** message was printed. As was explained above, the **FAILED** message might correspond to an empty test class that Fastest could not prune.



## 5.8 Timeout for Abstract Test Case Generation

There is one configuration variable, defined in the file `INSTALLDIR/lib/conf/fastest.conf`, that can be used to change the time that Fastest will spend in trying to find a test case for a given test class. The variable is called `SETLOG_TIMEOUT` and must be equal to some integer number in milliseconds. The default value is 10000, that is Fastest will try to find a test case for each test class for at most 10 seconds.

Every time this value is changed, Fastest must be restarted.

Fastest is delivered with the best value according to our experience.

## 5.9 Exploring and Saving the Results

The specification and the results of the work carried on with Fastest can be displayed or saved in  $\text{\LaTeX}$  format with the commands of the `show` family.

**Important!!** If Fastest terminates by any means, all data will be lost unless the user has saved it in files with one or more of the commands described in this section.

`showloadedops` prints the names of all the Z schemas that look like operation schemas. Fastest considers that a Z schema is an operation schema if it includes input or before state variables, on one hand, and output or after state variables, on the other. If an schema is the result of an schema expression, then all of them might be considered operations. For instance, if  $A == B \vee C$  and  $B$  and  $C$  are operation schemas, then `showloadedops` will print something like:

```
* A
* B
* C
```

However, the user should select only  $A$  as an operation to be tested since  $B$  and  $C$  will be considered when DNF is applied. Operation selection is explained in section 5.3.

`showtatics` prints a brief description of all the available testing tactics. A deeper explanation of them can be found in section 5.5 and its subsections.

The remaining `show` commands display and save the specification, testing trees, test classes, abstract test cases and values bound to identifiers declared in axiomatic definitions. In any case, command options must be entered in the order they are documented. Some commands feature the `-o` option that redirects the output to a file. This is the only way, so far, to save the results generated by Fastest. The output of most of these commands is  $\text{\LaTeX}$  mark-up. The following table summarizes these commands.

Command	Description	Options
<code>showaxdefs</code>	Displays the axiomatic definitions present in the specification in $\text{\LaTeX}$ mark-up.	<code>[-o &lt;file_name&gt;]</code> redirects the output to a file
<code>showaxdefvalues</code>	Displays the values bound, either automatically or manually, to identifiers declared in axiomatic definitions.	<code>[-o &lt;file_name&gt;]</code> Same as before.

Command	Description	Options
<code>shows</code>	Displays a given schema (it can be either any of the specification schema, test classes or abstract test cases) $\text{\LaTeX}$ mark-up.	<p><code>&lt;sch_name&gt;</code></p> <p>The name of the schema to be displayed.</p> <p><code>[-u &lt;unfold_order&gt;]</code></p> <p>Displays the result with more or less detail (basically it expands up to some level the included schema boxes). <code>-u -1</code> expands all the schemas.</p> <p><code>[-o &lt;file_name&gt;]</code></p> <p>Same as before.</p>
<code>shows -tca</code>	Displays all of the schemas corresponding to abstract test cases (of all testing trees) $\text{\LaTeX}$ mark-up.	<p><code>[-p &lt;op_name&gt;]</code></p> <p>Displays only the abstract test cases of operation schema <code>op_name</code>.</p> <p><code>[-u &lt;unfold_order&gt;]</code></p> <p>Same as before.</p> <p><code>[-o &lt;file_name&gt;]</code></p> <p>Same as before.</p>
<code>shows -tcl</code>	Displays all of the schemas corresponding to test classes (of all testing trees) $\text{\LaTeX}$ mark-up.	<p><code>[-p &lt;op_name&gt;]</code></p> <p>Displays only the abstract test classes of operation schema <code>op_name</code>.</p> <p><code>[-u &lt;unfold_order&gt;]</code></p> <p>Same as before.</p> <p><code>[-o &lt;file_name&gt;]</code></p> <p>Same as before.</p>

Command	Description	Options
<code>showspec</code>	Displays the entire specification $\text{\LaTeX}$ mark-up.	<p><code>[-u]</code> Same as <code>-u -1</code> before.</p> <p><code>[-o &lt;file_name&gt;]</code> Same as before.</p>
<code>showtt</code>	Displays all of the testing trees.	<p><code>[-p &lt;op_name&gt;]</code> Displays only the testing tree of operation schema <code>op_name</code>.</p> <p><code>[-o &lt;file_name&gt;]</code> Same as before.</p> <p><code>[-x]</code> Displays also the test classes that were pruned.</p>

## 5.10 How to Quit Fastest

**Important!!** Before leaving Fastest save your results. Fastest will not save anything by default and will not remember you that there is unsaved data.

To leave the program just type `quit` and press the return key.

## A Z Features Unsupported by Fastest

The following Z features are still unsupported by Fastest; the list is not exhaustive.

- Type synonyms are not supported.
- The hide (`\hide`, `\`) operator.  
Fastest will crash if the specification being loaded uses this operator.
- Schema names referenced in the predicate part of some schema.  
The referenced schemas will not be unfolded, thus severely reducing the effectiveness of both automatic pruning and abstract test case search. Test case design will still be quite meaningful.
- Variable substitution.  
Schema expressions such as  $A[a/b]$  where  $A$  is a schema and  $a$  and  $b$  are variables, are not supported. If  $B == A[a/b]$ , then  $B$  will not be recognized as an operation regardless of  $A$ .
- The following operators: **if** .  
The **if** clause will not be rewritten when DNF is calculated, as it should be.
- The L<sup>A</sup>T<sub>E</sub>X mark-up `\input`.  
Command `loadspect` will only load the specification explicitly present in the file passed as parameter. It will ignore any `\input` commands present in that file.
- Inductive types.  
Fastest is unable to find abstract test cases from test classes whose predicates include references to (non-constant) constructors defined in inductive types. Automatic pruning might not work correctly in this case.  
Though, enumerated types are fully supported.
- The  $\theta$  operator.  
However, Fastest should be able to find abstract test cases for test classes using variables whose type is a schema type.
- The Z sectioning system.  
Z sections are not recognized by Fastest.
- Generic definitions and generic schemas.  
This features are not supported although the user can perform test case design with specifications using them. Fastest will work as usual for those operation schemas that do not use generics.
- Schema composition and piping.  
Schemas defined by these operators will not be recognized as operations by Fastest, thus making it impossible for the user to “test” them.  
Section 4.4 might give some light on how to deal with this limitation.

## B Test Classes Generated for *KeepMaxReading*

The following schema boxes represent the test classes generated for the operation *KeepMaxReading*. In the framework developed in [Sto93] each test class is described as a Z schema. This is important because only one notation is necessary to describe the specification and the test results.

<i>KeepMaxReading_VIS</i>
$smax : SENSOR \rightarrow \mathbb{Z}$
$s? : SENSOR$
$r? : \mathbb{Z}$

<i>KeepMaxReading_DNF_1</i>
$smax : SENSOR \rightarrow \mathbb{Z}$
$s? : SENSOR$
$r? : \mathbb{Z}$
$s? \in \text{dom } smax$
$smax\ s? < r?$

<i>KeepMaxReading_SP_1</i>
$smax : SENSOR \rightarrow \mathbb{Z}$
$s? : SENSOR$
$r? : \mathbb{Z}$
$s? \in \text{dom } smax$
$smax\ s? < r?$
$smax\ s? < 0$
$r? < 0$

<i>KeepMaxReading_SP_2</i>
$smax : SENSOR \rightarrow \mathbb{Z}$
$s? : SENSOR$
$r? : \mathbb{Z}$
$s? \in \text{dom } smax$
$smax\ s? < r?$
$smax\ s? < 0$
$r? = 0$

<i>KeepMaxReading_SP_3</i>
$smax : SENSOR \rightarrow \mathbb{Z}$
$s? : SENSOR$
$r? : \mathbb{Z}$
$s? \in \text{dom } smax$
$smax\ s? < r?$
$smax\ s? < 0$
$r? > 0$

<i>KeepMaxReading_SP_4</i>
$smax : SENSOR \rightarrow \mathbb{Z}$
$s? : SENSOR$
$r? : \mathbb{Z}$
$s? \in \text{dom } smax$
$smax\ s? < r?$
$smax\ s? = 0$
$r? > 0$

<i>KeepMaxReading_SP_5</i>
$smax : SENSOR \rightarrow \mathbb{Z}$
$s? : SENSOR$
$r? : \mathbb{Z}$
$s? \in \text{dom } smax$
$smax\ s? < r?$
$smax\ s? > 0$
$r? > 0$

<i>KeepMaxReading_DNF_2</i>
$smax : SENSOR \rightarrow \mathbb{Z}$
$s? : SENSOR$
$r? : \mathbb{Z}$
$s? \notin \text{dom } smax$

<i>KeepMaxReading_SP_6</i>
$smax : SENSOR \rightarrow \mathbb{Z}$
$s? : SENSOR$
$r? : \mathbb{Z}$
$s? \notin \text{dom } smax$
$smax\ s? < 0$
$r? < 0$

<i>KeepMaxReading_SP_7</i>
$smax : SENSOR \rightarrow \mathbb{Z}$
$s? : SENSOR$
$r? : \mathbb{Z}$
$s? \notin \text{dom } smax$
$smax\ s? < 0$
$r? = 0$

*KeepMaxReading\_SP\_8* \_\_\_\_\_

$smax : SENSOR \rightarrow \mathbb{Z}$   
 $s? : SENSOR$   
 $r? : \mathbb{Z}$

$s? \notin \text{dom } smax$   
 $smax\ s? < 0$   
 $r? > 0$

*KeepMaxReading\_SP\_9* \_\_\_\_\_

$smax : SENSOR \rightarrow \mathbb{Z}$   
 $s? : SENSOR$   
 $r? : \mathbb{Z}$

$s? \notin \text{dom } smax$   
 $smax\ s? = 0$   
 $r? > 0$

*KeepMaxReading\_SP\_10* \_\_\_\_\_

$smax : SENSOR \rightarrow \mathbb{Z}$   
 $s? : SENSOR$   
 $r? : \mathbb{Z}$

$s? \notin \text{dom } smax$   
 $smax\ s? > 0$   
 $r? > 0$

*KeepMaxReading\_DNF\_3* \_\_\_\_\_

$smax : SENSOR \rightarrow \mathbb{Z}$   
 $s? : SENSOR$   
 $r? : \mathbb{Z}$

$s? \in \text{dom } smax$   
 $r? \leq smax\ s?$

*KeepMaxReading\_SP\_11* \_\_\_\_\_

$smax : SENSOR \rightarrow \mathbb{Z}$   
 $s? : SENSOR$   
 $r? : \mathbb{Z}$

$s? \in \text{dom } smax$   
 $r? \leq smax\ s?$   
 $smax\ s? < 0$   
 $r? < 0$

*KeepMaxReading\_SP\_12* \_\_\_\_\_

$smax : SENSOR \rightarrow \mathbb{Z}$   
 $s? : SENSOR$   
 $r? : \mathbb{Z}$

$s? \in \text{dom } smax$   
 $r? \leq smax\ s?$   
 $smax\ s? < 0$   
 $r? = 0$

*KeepMaxReading\_SP\_13* \_\_\_\_\_

$smax : SENSOR \rightarrow \mathbb{Z}$   
 $s? : SENSOR$   
 $r? : \mathbb{Z}$

$s? \in \text{dom } smax$   
 $r? \leq smax\ s?$   
 $smax\ s? < 0$   
 $r? > 0$

*KeepMaxReading\_SP\_14* \_\_\_\_\_

$smax : SENSOR \rightarrow \mathbb{Z}$   
 $s? : SENSOR$   
 $r? : \mathbb{Z}$

$s? \in \text{dom } smax$   
 $r? \leq smax\ s?$   
 $smax\ s? = 0$   
 $r? > 0$

*KeepMaxReading\_SP\_15* \_\_\_\_\_

$smax : SENSOR \rightarrow \mathbb{Z}$   
 $s? : SENSOR$   
 $r? : \mathbb{Z}$

$s? \in \text{dom } smax$   
 $r? \leq smax\ s?$   
 $smax\ s? > 0$   
 $r? > 0$

## C Standard Partitions

The following standard partitions are included by default in the file `lib/conf/stdpartition.spf`. The user can erase or modify these partitions and define new ones as well by simply editing the file. Fastest needs to be restarted so it is notified of changes.

### C.1 Sets

**Standard partition** for expressions of the form  $S \cup T$

$$\begin{aligned} S &= \{\}, T = \{\} \\ S &= \{\}, T \neq \{\} \\ S &\neq \{\}, T = \{\} \\ S &\neq \{\}, T \neq \{\}, S \cap T = \{\} \\ S &\neq \{\}, T \neq \{\}, S \subset T \\ S &\neq \{\}, T \neq \{\}, T \subset S \\ S &\neq \{\}, T \neq \{\}, T = S \\ S &\neq \{\}, T \neq \{\}, (S \cap T) \neq \{\}, \neg (S \subseteq T), \neg (T \subseteq S), T \neq S \end{aligned}$$

**Standard partition** for expressions of the form  $S \cap T$

$$\begin{aligned} S &= \{\}, T = \{\} \\ S &= \{\}, T \neq \{\} \\ S &\neq \{\}, T = \{\} \\ S &\neq \{\}, T \neq \{\}, S \cap T = \{\} \\ S &\neq \{\}, T \neq \{\}, S \subset T \\ S &\neq \{\}, T \neq \{\}, T \subset S \\ S &\neq \{\}, T \neq \{\}, T = S \\ S &\neq \{\}, T \neq \{\}, (S \cap T) \neq \{\}, \neg (S \subseteq T), \neg (T \subseteq S), T \neq S \end{aligned}$$

**Standard partition** for expressions of the form  $S \setminus T$

$$\begin{aligned} S &= \{\}, T = \{\} \\ S &= \{\}, T \neq \{\} \\ S &\neq \{\}, T = \{\} \\ S &\neq \{\}, T \neq \{\}, S \cap T = \{\} \\ S &\neq \{\}, T \neq \{\}, S \subset T \\ S &\neq \{\}, T \neq \{\}, T \subset S \\ S &\neq \{\}, T \neq \{\}, T = S \\ S &\neq \{\}, T \neq \{\}, (S \cap T) \neq \{\}, \neg (S \subseteq T), \neg (T \subseteq S), T \neq S \end{aligned}$$

**Standard partition** for expressions of the form  $x \notin A$

$$\begin{aligned} A &= \{\} \\ A &\neq \{\} \end{aligned}$$

**Standard partition** for expressions of the form  $x \in A$

$$\begin{aligned} A &= \{x\} \\ A &\neq \{x\}, x \in A \end{aligned}$$

**Standard partition** for expressions of the form  $\#A$

$$\begin{aligned} \#A &= 0 \\ \#A &= 1 \\ \#A &> 1 \end{aligned}$$

## C.2 Integers

**Standard partition** for expressions of the form  $n < m$

$$A < 0, B < 0$$

$$A < 0, B = 0$$

$$A < 0, B > 0$$

$$A = 0, B > 0$$

$$A > 0, B > 0$$

**Standard partition** for expressions of the form  $n \leq m$

$$A < 0, B < 0, A < B$$

$$A < 0, B < 0, A = B$$

$$A < 0, B = 0$$

$$A < 0, B > 0$$

$$A = 0, B = 0$$

$$A = 0, B > 0$$

$$A > 0, B > 0, A < B$$

$$A > 0, B > 0, A = B$$

**Standard partition** for expressions of the form  $n > m$

$$A < 0, B < 0$$

$$A < 0, B = 0$$

$$A = 0, B < 0$$

$$A > 0, B = 0$$

$$A > 0, B > 0$$

**Standard partition** for expressions of the form  $n \geq m$

$$A < 0, B < 0, A > B$$

$$A < 0, B < 0, A = B$$

$$A = 0, B < 0$$

$$A = 0, B = 0$$

$$A > 0, B < 0$$

$$A > 0, B = 0$$

$$A > 0, B > 0, A > B$$

$$A > 0, B > 0, A = B$$

**Standard partition** for expressions of the form  $n = m$

$$A < 0, B < 0$$

$$A = 0, B = 0$$

$$A > 0, B > 0$$

**Standard partition** for expressions of the form  $n \neq m$

$$n < 0, m < 0$$

$$n < 0, m = 0$$

$$n < 0, m > 0$$

$$n = 0, m < 0$$

$$n = 0, m > 0$$

$$n > 0, m < 0$$

$$n > 0, m = 0$$

$$n > 0, m > 0$$



**Standard partition** for expressions of the form  $n + m$

$n < 0, m < 0, n < m$   
 $n < 0, m < 0, n = m$   
 $n < 0, m < 0, n > m$   
 $n < 0, m = 0$   
 $n < 0, m > 0$   
 $n = 0, m < 0$   
 $n = 0, m = 0$   
 $n = 0, m > 0$   
 $n > 0, m > 0, n < m$   
 $n > 0, m > 0, n = m$   
 $n > 0, m > 0, n > m$

### C.3 Relations

**Standard partition** for expressions of the form  $R \oplus G$

$R = \{\}, G = \{\}$   
 $R = \{\}, G \neq \{\}$   
 $R \neq \{\}, G = \{\}$   
 $R \neq \{\}, G \neq \{\}, \text{dom } R = \text{dom } G$   
 $R \neq \{\}, G \neq \{\}, \text{dom } G \subset \text{dom } R$   
 $R \neq \{\}, G \neq \{\}, (\text{dom } R \cap \text{dom } G) = \{\}$   
 $R \neq \{\}, G \neq \{\}, \text{dom } R \subset \text{dom } G$   
 $R \neq \{\}, G \neq \{\}, (\text{dom } R \cap \text{dom } G) \neq \{\}, \neg (\text{dom } G \subseteq \text{dom } R), \neg (\text{dom } R \subseteq \text{dom } G)$

**Standard partition** for expressions of the form  $S \triangleleft R$

$R = \{\}$   
 $R \neq \{\}, S = \{\}$   
 $R \neq \{\}, S = \text{dom } R$   
 $R \neq \{\}, S \neq \{\}, S \subset \text{dom } R$   
 $R \neq \{\}, S \neq \{\}, S \cap \text{dom } R = \{\}$   
 $R \neq \{\}, S \cap \text{dom } R \neq \{\}, \text{dom } R \subset S$   
 $R \neq \{\}, S \cap \text{dom } R \neq \{\}, \neg (\text{dom } R \subseteq S), \neg (S \subseteq \text{dom } R)$

**Standard partition** for expressions of the form  $S \triangleleft R$

$R = \{\}$   
 $R \neq \{\}, S = \{\}$   
 $R \neq \{\}, S = \text{dom } R$   
 $R \neq \{\}, S \neq \{\}, S \subset \text{dom } R$   
 $R \neq \{\}, S \neq \{\}, S \cap \text{dom } R = \{\}$   
 $R \neq \{\}, S \cap \text{dom } R \neq \{\}, \text{dom } R \subset S$   
 $R \neq \{\}, S \cap \text{dom } R \neq \{\}, \neg (\text{dom } R \subseteq S), \neg (S \subseteq \text{dom } R)$

**Standard partition** for expressions of the form  $R \triangleright S$

$$\begin{aligned}
& R = \{\} \\
& R \neq \{\}, S = \{\} \\
& R \neq \{\}, S = \text{ran } R \\
& R \neq \{\}, S \neq \{\}, S \subset \text{ran } R \\
& R \neq \{\}, S \neq \{\}, S \cap \text{ran } R = \{\} \\
& R \neq \{\}, S \cap \text{ran } R \neq \{\}, \text{ran } R \subset S \\
& R \neq \{\}, S \cap \text{ran } R \neq \{\}, \neg (\text{ran } R \subseteq S), \neg (S \subseteq \text{ran } R)
\end{aligned}$$

## C.4 Sequences

**Standard partition** for expressions of the form  $s \frown t$

The case  $\# \text{ran } s > \#s$  is impossible.

$$\begin{aligned}
& s = \langle \rangle, t = \langle \rangle \\
& s \neq \langle \rangle, t = \langle \rangle \\
& s = \langle \rangle, t \neq \langle \rangle \\
& s \neq \langle \rangle, t \neq \langle \rangle, \#s < \#t, \# \text{ran } s < \#s, \# \text{ran } t < \#t \\
& s \neq \langle \rangle, t \neq \langle \rangle, \#s < \#t, \# \text{ran } s = \#s, \# \text{ran } t < \#t \\
& s \neq \langle \rangle, t \neq \langle \rangle, \#s < \#t, \# \text{ran } s = \#s, \# \text{ran } t = \#t \\
& s \neq \langle \rangle, t \neq \langle \rangle, \#s < \#t, \# \text{ran } s < \#s, \# \text{ran } t = \#t \\
& s \neq \langle \rangle, t \neq \langle \rangle, \#s = \#t, \# \text{ran } s < \#s, \# \text{ran } t < \#t \\
& s \neq \langle \rangle, t \neq \langle \rangle, \#s = \#t, \# \text{ran } s = \#s, \# \text{ran } t < \#t \\
& s \neq \langle \rangle, t \neq \langle \rangle, \#s = \#t, \# \text{ran } s = \#s, \# \text{ran } t = \#t \\
& s \neq \langle \rangle, t \neq \langle \rangle, \#s = \#t, \# \text{ran } s < \#s, \# \text{ran } t = \#t \\
& s \neq \langle \rangle, t \neq \langle \rangle, \#s > \#t, \# \text{ran } s < \#s, \# \text{ran } t < \#t \\
& s \neq \langle \rangle, t \neq \langle \rangle, \#s > \#t, \# \text{ran } s = \#s, \# \text{ran } t < \#t \\
& s \neq \langle \rangle, t \neq \langle \rangle, \#s > \#t, \# \text{ran } s = \#s, \# \text{ran } t = \#t \\
& s \neq \langle \rangle, t \neq \langle \rangle, \#s > \#t, \# \text{ran } s < \#s, \# \text{ran } t = \#t
\end{aligned}$$

**Standard partition** for expressions of the form  $\text{squash}(f)$

$$\begin{aligned}
& f = \{\} \\
& f \neq \{\}, 1 \in \text{dom } f, \text{dom } f \subset 1 \dots \#f - 1, \# \text{ran } f < \#f \\
& f \neq \{\}, 1 \in \text{dom } f, \text{dom } f \subset 1 \dots \#f - 1, \# \text{ran } f = \#f \\
& f \neq \{\}, \text{dom } f \subset 2 \dots \#f - 1, \# \text{ran } f < \#f \\
& f \neq \{\}, \text{dom } f \subset 2 \dots \#f - 1, \# \text{ran } f = \#f \\
& f \neq \{\}, \#f \in \text{dom } f, \text{dom } f \subset 2 \dots \#f, \# \text{ran } f < \#f \\
& f \neq \{\}, \#f \in \text{dom } f, \text{dom } f \subset 2 \dots \#f, \# \text{ran } f = \#f \\
& f \neq \{\}, \text{dom } f = 1 \dots \#f, \# \text{ran } f < \#f \\
& f \neq \{\}, \text{dom } f = 1 \dots \#f, \# \text{ran } f = \#f \\
& f \neq \{\}, 1 \in \text{dom } f, \#f \notin \text{dom } f, \neg (\text{dom } f \subseteq 1 \dots \#f), \# \text{ran } f < \#f \\
& f \neq \{\}, 1 \in \text{dom } f, \#f \notin \text{dom } f, \neg (\text{dom } f \subseteq 1 \dots \#f), \# \text{ran } f = \#f \\
& f \neq \{\}, 1 \notin \text{dom } f, \#f \notin \text{dom } f, \text{dom } f \cap (1 \dots \#f) \neq \{\}, \neg (\text{dom } f \subseteq 1 \dots \#f), \# \text{ran } f < \#f \\
& f \neq \{\}, 1 \notin \text{dom } f, \#f \notin \text{dom } f, \text{dom } f \cap (1 \dots \#f) \neq \{\}, \neg (\text{dom } f \subseteq 1 \dots \#f), \# \text{ran } f = \#f \\
& f \neq \{\}, 1 \notin \text{dom } f, \#f \in \text{dom } f, \neg (\text{dom } f \subseteq 1 \dots \#f), \# \text{ran } f < \#f \\
& f \neq \{\}, 1 \notin \text{dom } f, \#f \in \text{dom } f, \neg (\text{dom } f \subseteq 1 \dots \#f), \# \text{ran } f = \#f \\
& f \neq \{\}, \text{dom } f \cap (1 \dots \#f) = \{\}, \# \text{ran } f < \#f \\
& f \neq \{\}, \text{dom } f \cap (1 \dots \#f) = \{\}, \# \text{ran } f = \#f
\end{aligned}$$

## References

- [Abr96] J.-R. Abrial. *The B-book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996.
- [BCK03] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice, Second Edition*. Addison-Wesley Professional, April 2003.
- [BGM91] Gilles Bernot, Marie Claude Gaudel, and Bruno Marre. Software testing based on formal specifications: a theory and a tool. *Softw. Eng. J.*, 6(6):387–405, 1991.
- [Bow] Jonathan Bowen. Formal methods. <http://vl.fimnet.info/>.
- [Bro95] Frederick P. Brooks, Jr. *The mythical man-month (anniversary ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
- [CAF<sup>+</sup>14] Maximiliano Cristiá, Pablo Albertengo, Claudia S. Frydman, Brian Plüss, and Pablo Rodríguez Monetti. Tool support for the Test Template Framework. *Softw. Test., Verif. Reliab.*, 24(1):3–37, 2014.
- [DBA<sup>+</sup>01] R. Dupuis, P. Bourque, A. Abran, J. W. Moore, and L. L. Tripp. The SWEBOK Project: Guide to the software engineering body of knowledge, May 2001. Stone Man Trial Version 1.00, <http://www.swebok.org/> [01/12/2003].
- [GGSV02] Wolfgang Grieskamp, Yuri Gurevich, Wolfram Schulte, and Margus Veanes. Generating finite state machines from abstract state machines. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 112–122, New York, NY, USA, 2002. ACM.
- [GJM03] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of software engineering (2nd ed.)*. Prentice Hall, 2003.
- [HB99] M.G. Hinchey and J.P. Bowen. *Industrial-strength formal methods in practice*. Formal approaches to computing and information technology. Springer, 1999.
- [HBB<sup>+</sup>09] Robert M. Hierons, Kirill Bogdanov, Jonathan P. Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, Gerald Lüttgen, Anthony J. H. Simons, Sergiy Vilkomir, Martin R. Woodward, and Hussein Zedan. Using formal specifications to support testing. *ACM Comput. Surv.*, 41(2):1–76, 2009.
- [ISO02] ISO. Information Technology – Z Formal Specification Notation – Syntax, Type System and Semantics. Technical Report ISO/IEC 13568, International Organization for Standardization, 2002.
- [Lam02] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
- [LPU02] Bruno Legeard, Fabien Peureux, and Mark Utting. A Comparison of the BTT and TTF Test-Generation Methods. In *ZB '02: Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B*, pages 309–329, London, UK, 2002. Springer-Verlag.

- [McC04] Steve McConnell. *Code Complete, Second Edition*. Microsoft Press, Redmond, WA, USA, 2004.
- [MFC<sup>+</sup>09] Bertrand Meyer, Arno Fiva, Ilinca Ciupa, Andreas Leitner, Yi Wei, and Emmanuel Stapf. Programs that test themselves. *Computer*, 42:46–55, September 2009.
- [NSV<sup>+</sup>08] Arilo Dias Neto, Rajesh Subramanyan, Marlon Vieira, Guilherme Horta Travassos, and Forrest Shull. Improving evidence about software technologies: A look at model-based testing. *IEEE Softw.*, 25(3):10–13, 2008.
- [Pff01] Shari Lawrence Pfleeger. *Software Engineering: Theory and Practice*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.
- [RTI02] RTI. The economic impacts of inadequate infrastructure for software testing. Planning Report 02-3, National Institute of Standards and Technology, Gaithersburg, MD, May 2002.
- [SC96] P. Stocks and D. Carrington. A Framework for Specification-Based Testing. *IEEE Transactions on Software Engineering*, 22(11):777–793, November 1996.
- [Sto93] P. Stocks. *Applying Formal Methods to Software Testing*. PhD thesis, Department of Computer Science, University of Queensland, 1993.
- [UL06] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.