Implementing and Applying the Stocks-Carrington Framework for Model-Based Testing

Maximiliano Cristiá[1][2][3] and Pablo Rodríguez Monetti[1][2]

[1] Flowgate Consulting
[2] FCEIA, Universidad Nacional de Rosario
[3] CIFASIS-CONICET
{mcristia,prodriguez}@flowgate.net

Abstract. In this paper we describe the functional features and the architecture of a tool implementing the Stocks-Carrington framework (TTF) for model based testing (MBT). The resulting prototype, called Fastest, makes it easy to generate test cases from Z specifications. We not only apply the ideas of the referred framework but we also use a technique based on finite models to find test cases, which has proved to increase the level of automation during the whole testing process. The paper also discusses problems and challenges that have appeared during the development of the tool, and introduces real case studies and an analysis of the results obtained so far.

1 Introduction

Current industrial practice in software testing is mostly manual: test template[4] definition, test case derivation and the determination of success or failure are all manual, tedious, resource consuming and error prone activities. These activities make a poor use of software engineers' skills, changing analytical tasks for a repetitive manual work. Within this picture many software development companies seldom perform a serious testing of their programs. In this paper we introduce a tool, called Fastest [1], which automates test suite definition and test case derivation for unit testing, by applying the Test Template Framework (TTF) proposed by Stocks and Carrington [2–4] for model-based testing (MBT).

The main contributions of the paper are: (a) a flexible, efficient and automatic implementation of the TTF which, as far as we have investigated, does not exist yet; (b) the implementation of a technique to automatically derive abstract test cases from test objectives, which was not originally proposed by Stocks and Carrington; and (c) an analysis of the application of Fastest to two industrial-strength case studies and other toy examples.

Since this paper is about MBT we will shortly introduce this technique, for a deeper presentation the reader may consult [5]. In order to test a program

[4] Test templates are also called *test classes* or *test objectives*; we will use all of them as synonymous.

P it is necessary to have a formal specification Op of it and follow the process depicted in Fig. 1. From Op, abstract test cases (t) are derived by executing the conceptual step named *gen*. Abstract test cases are abstract since they are written in the same language than Op. Then each t is refined into a concrete test case (x), i.e. in a test case written in the same language than P. The *exec* step represents the actual execution of P with input x, which produces an output $P(x)$. This output is abstracted to the specification level when the *abs* step is executed. The specification is used again to verify whether the test has uncovered and error or not, during the final step *compr*. The abstract pair (x, y) is conveniently substituted in Op: if it reduces to *true* then no error was discovered, and if it reduces to *false* an error was found.

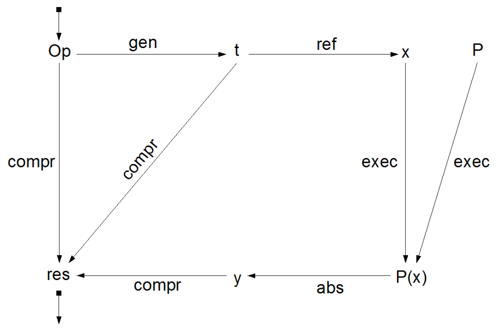

Fig. 1. A general description of the MBT process.

All of the steps of Fig. 1 can be mechanized to a great extent. In this article we introduce a specific implementation of the *gen* step based on the TTF. It is worth to mention that, although the Z notation is one of the most used formal notations in industry and that the TTF is a straightforward and powerful MBT technique for Z, there is no implementation of the Stocks and Carrington framework as automatic as ours.

The paper is structured as follows. The next section presents a brief introduction to the TTF. In Sect. 3 we show the functional and architectural features of Fastest, while a representative example is run in Sect. 4. Section 5 presents the results of applying Fastest to some industrial cases studies and to some toy examples, as well. Our work is compared to similar approaches in Sect. 6. Finally, Sect. 7 describe our conclusions and future work.

2 The Test Template Framework

In this section we introduce MBT from the perspective of Stocks and Carrington, without mention any particular implementation or tool. Precisely, in Sect. 3 we introduce our implementation which also refines the framework.

Phil Stocks and David Carrington introduced in [2–4] a formal framework to conduct model-based testing of Z specifications [6], including a rigorous and disciplined technique for defining and structuring abstract test templates and cases. They also proposed new testing tactics[5] specifically suited to the Z notation. According to them, testing tactics are the mechanisms that must be used to partition the input space to build a so called testing tree.

We make use of an example to explain the TTF, but we assume the reader is fluent in the Z notation.

2.1 A Simple Pool of Sensors

The following Z model describes a simple pool of sensors which records the highest reading of each sensor. The *KMR* operation[6] takes as input a sensor ID, $s?$, and a reading, $r?$, of it. If $s?$ is a valid ID and if $r?$ is greater than the current reading of $s?$, *KMR* replaces the current reading for $r?$. If some condition does not hold, then the operation fails and nothing is changed.

$$[SENSOR]$$
$$MaxReadings == [smax : SENSOR \pfun \mathbb{Z}]$$

$$\begin{array}{|l}
\multicolumn{1}{l}{KMROk} \\ \hline
\Delta MaxReadings \\
s? : SENSOR;\ r? : \mathbb{Z} \\ \hline
s? \in \operatorname{dom} smax \\
smax\ s? < r? \\
smax' = smax \oplus \{s? \mapsto r?\} \\ \hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{KMRE2} \\ \hline
\Xi MaxReadings \\
s? : SENSOR;\ r? : \mathbb{Z} \\ \hline
s? \in \operatorname{dom} smax \\
r? \leq smax\ s? \\ \hline
\end{array}$$

$$KMRE1 == [\Xi MaxReadings;\ s? : SENSOR \mid s? \notin \operatorname{dom} smax]$$
$$KMR == KMROk \lor KMRE1 \lor KMRE2$$

2.2 Testing Tactics, Test Classes and Testing Trees

The TTF starts by defining, for each Z operation, the *input space* (*IS*) and the *valid input space* (*VIS*). The *IS* is the set defined by all of the possible values of the input and state variables of the operation. For instance, the *IS* of *KMR* is:

[5] Here we preferred testing tactic instead of the original term *testing strategy* because we believe the former has a narrow scope than the last.

[6] *KMR* stands for *KeepMaxReadings*.

$$IS == [smax : SENSOR \pfun \mathbb{Z};\ s? : SENSOR;\ r? : \mathbb{Z}]$$

In turn, the *VIS* is the subset of *IS* for which the operation is defined. The *VIS* of *KMR* is equal to its *IS* since the operation is total. More formally, the *VIS* of an operation *Op* can be defined as follows:

$$VIS_{Op} == [IS_{Op} \mid \mathrm{pre}\ Op]$$

Stocks and Carrington suggest to divide the *VIS* into equivalence classes, called *test classes*, by applying one or more *testing tactics*. The test classes obtained in this way can be further subdivided into more test classes by applying other testing tactics. This procedure can continue until either the engineer considers that the test classes are reasonable small or there is a reasonable number or each of the functional alternatives is covered by only one test class. Within the TTF all these test classes are represented as a *testing tree* as shown in Fig. 3. The authors indicate that it is convenient for the testing tactics to produce a mathematical partition of either the *VIS* or a test class. In this way, test classes are indeed equivalence classes in the sense that if the program fails for a particular element it should fail for any other element of the class. Test cases are taken only from the leafs of the testing as we will see shortly.

Within the TTF the authors defined a number of testing tactics that together provide a sound method for calculating tests objectives. Furthermore, they propose that new tactics should be added for particular projects, systems, requirements, etc. Then, any tool implementing the TTF should enable the inclusion and parametrization of testing tactics. Two of the testing tactics proposed within the TTF are the following:

- *Disjunctive Normal Form (DNF).* By applying this tactic the predicate of the operation is written in DNF and the *VIS* or a test class is partitioned in as many test classes as terms has the predicate. The precondition of each term is conjoined to the predicate of the *VIS* or the test class being divided.
- *Standard Partitions (SP).* This tactic uses the mathematical operators appearing in the predicate to generate more test objectives. A *standard partition* is a partition of the domain of a given mathematical operator in sets called *sub-domains*. In practice, standard partitions are expressed as conditions that must hold of the operator's operands. These conditions are used to divide a test class. Figure 2 shows the standard partition for the $<$ operator.

1. $a < 0,\ b < 0$	4. $a = 0,\ b < 0$	7. $a > 0,\ b < 0$
2. $a < 0,\ b = 0$	5. $a = 0,\ b = 0$	8. $a > 0,\ b = 0$
3. $a < 0,\ b > 0$	6. $a = 0,\ b > 0$	9. $a > 0,\ b > 0$

Fig. 2. Standard partition for $a < b$.

We will apply both tactics to *KMR*. First we apply DNF to the *VIS* and then Standard Partitions is applied to the expression $smax\ s? < r?$ of *KMROk*. The resulting testing tree is shown in Fig. 3. Some of the nodes of the testing tree are shown below. It is worth to mention that the predicates of the objectives in a given level contain the predicates of the upper level. This is the reason for which test cases are drawn only from the leafs.

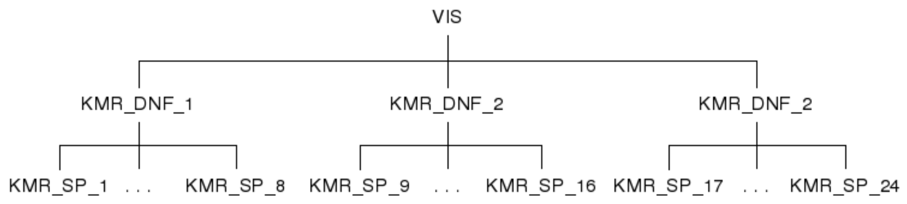

Fig. 3. Testing tree of *KMR*.

KMR_DNF_1

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

$s? \in \operatorname{dom} smax$
$smax\ s? < r?$

KMR_SP_2

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

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

KMR_SP_1

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

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

KMR_SP_9

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

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

$$
\begin{array}{|l}
\multicolumn{1}{l}{KMR\_SP\_19} \\
\hline
smax : SENSOR \pfun \mathbb{Z} \\
s? : SENSOR \\
r? : \mathbb{Z} \\
\hline
s? \in \operatorname{dom} smax \\
r? \leq smax\ s? \\
smax\ s? < 0 \\
r? < 0 \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{KMR\_SP\_20} \\
\hline
smax : SENSOR \pfun \mathbb{Z} \\
s? : SENSOR \\
r? : \mathbb{Z} \\
\hline
s? \in \operatorname{dom} smax \\
r? \leq smax\ s? \\
smax\ s? < 0 \\
r? = 0 \\
\hline
\end{array}
$$

The second step of the TTF methodology suggest to prune the testing tree. In effect, some test objectives are empty because they are contradictions. In these cases it is impossible to find abstract test cases. For instance, class KMR_SP_20 must be pruned, among others. Stocks and Carrington do not give a recipe on how this can be automated. In Sect. 3.6 we give a very short description of a technique that we are currently implementing in Fastest.

Finally, the engineer has to choose at least one element for each of the remaining leafs of the testing tree. These are the abstract test cases. Here, we can see one of the main benefits of the TTF: the model, the test classes and the test cases are all expressed in the same notation. For example the following schema boxes represent abstract test cases of the corresponding test objectives. Note that the method naturally provides traceability between objectives and abstract test cases by using schema inclusion. Note that within the TTF a test case is a sort of assignment of constant values to state and input variables, rather than a sequence of operations leading to the desired state, as is suggested by other approaches [7–9].

$$
\begin{array}{|l}
\multicolumn{1}{l}{KMR\_SP\_1\_TCASE} \\
\hline
KMR\_SP\_1 \\
\hline
r? = \; -1 \\
smax = \{(sensor0, -2)\} \\
s? = sensor0 \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{KMR\_SP\_6\_TCASE} \\
\hline
KMR\_SP\_6 \\
\hline
r? = 1 \\
smax = \{(sensor0, 0)\} \\
s? = sensor0 \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{KMR\_SP\_2\_TCASE} \\
\hline
KMR\_SP\_2 \\
\hline
r? = 0 \\
smax = \{(sensor0, -1)\} \\
s? = sensor0 \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{KMR\_SP\_9\_TCASE} \\
\hline
KMR\_SP\_9 \\
\hline
r? = 2 \\
smax = \{(sensor0, 1)\} \\
s? = sensor0 \\
\hline
\end{array}
$$

Although the TTF suggests that the selection of abstract test cases from test classes is a rather easy task, the authors do not give any clue on how to automate this step. In Sect. 3.7 we give such method.

3 Fastest: an Implementation of the TTF

The goal of this section is to describe both the functional and architectural features of Fastest. Also we show some design and implementation details that we believe contribute to the development of MBT tools. Since the Community Z Tools framework [10, 11] is a key component of Fastest we show how it was integrated into our tool.

3.1 Conceptual Description

As shown in Fig. 4, Fastest receives a Z specification in LaTeX format using the CZT package. The Z specification must verify the ISO standard [12]. The specification is transformed into an internal representation more amenable to parsing and static analysis. Then, the user has to enter a list of the operations to test as well as the tactics to apply to each of them. In a third step Fastest generates the testing tree of each operation. After the trees are generated, the user can browse them and their test classes, and he can prune any node. Once the user is done with pruning, he can instruct Fastest to find one abstract test case for each leaf in all the test trees. Although the method to find abstract test cases has proved to be quite automatic, it is worth to say that it does not guarantee to find abstract test cases for all test objectives, as we will see in a following subsection. The user can export all the results –test classes and abstract test cases– in LaTeX format.

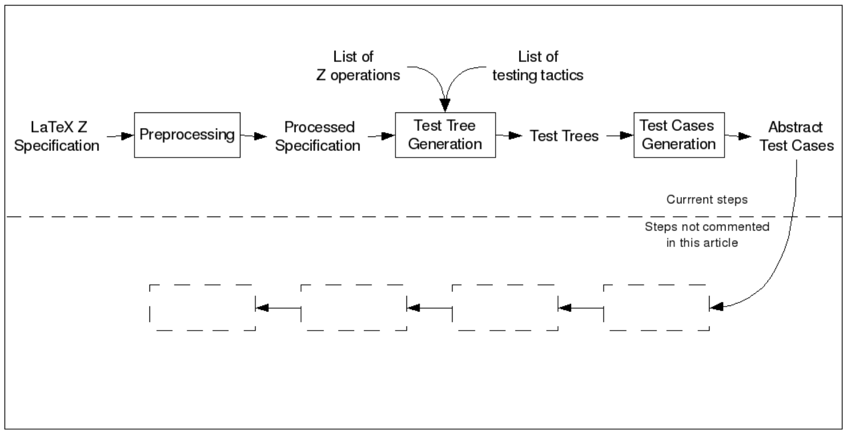

Fig. 4. Fastest's testing process. The dotted line divides the steps reported in this paper from those not included here.

3.2 The Architecture of Fastest

Fastest was envisioned as a client-server application [13]. The main reason for thinking in a distributed system came from the realization that calculating abstract test cases from test objectives in large projects could be a hard computing problem, but highly parallelizable as well –we will come back to this point in Sect. 3.7. Then, we thought in an scalable application using the idle computer power present in a corporate network. On the other side, since testing a large application is usually carried on by a group of testers, a client-server architecture allows the team to work concurrently from different workstations by running a client process per tester. However, in such a large project there is shared information –such as the definition or parametrization of some testing tactics, test cases already calculated, theorems that help to prune testing trees, etc.– that all the clients and servers should be able to access. Hence, a typical Fastest installation has a data server that is know to all other processes, some client processes and a number of testing servers.

The client is organized following the Implicit Invocation architectural style [14]. The main architectural invariant of this style is that the components that announce events do not know which other components will react to them. A consequence of this property is the possibility to add, remove and change components without affecting the other [15]. For instance, it would be quite easy to add a component that is called whenever a new test case is calculated, that stores it on disc or that refines it to a given programming language.

The client has a simple text-based user interface from which the user can issued commands and read their output. All the components of this architecture were implemented as Java programs.

3.3 The CZT Framework

The Community Z Tools (CZT) project [10, 11] is an open source Java framework created in 2003 with the goal of building a toolkit for the Z notation or its dialects. These tools include editors, parsers, type-checkers and so on, to work with Z specifications written in LaTeX, Unicode or XML. The following CZT services have been used in Fastest:

– CZT Parser and the Annotated Ayntax Trees (AST)
– CZT Type-checker
– CZT ZLive specification animator

Although we were greatly benefited from all these services, perhaps the most important was ZLive. The ZLive project provides an evaluator of Z expressions and predicates. In other words, ZLive takes a Z model and values for the input and state variables and calculates the values of the output and next state variables –it is worth to mention that ZLive does not cover the whole Z notation yet. We used this facility to calculate test cases from test templates as is described in Sect. 3.7.

3.4 Testing Tactics in Fastest

As we have said early, the first step of the TTF is to apply some testing tactics to each operation of the Z model. Fastest features a user command (`addtactic`) that allows the user to build the list of tactics to be applied to each operation, by entering the name of an operation, the testing tactic name and a few more parameters depending on the tactic –for more details on the syntax of user commands go to Sect. 4. The current version implements DNF, Standard Partitions and a new tactic proposed by us called Free Types (FT). This tactic works on variables of a free (enumerated) type, partitioning the *VIS* or the test class in as many classes as elements has the type. More formally, if the *VIS* of a given operation declares $v : T$ where $T ::= V_1 \mid \ldots \mid V_n$, then the class is partitioned in n templates where each of them has a predicate of the form $v = V_i$. This tactic is useful since enumerated types usually codify specific states, operational modes, etc. Hence, by applying it there will be at least one test case for each of those situations.

Since Standard Partitions is a general tactic that can be applied to a number of mathematical operators, Fastest has a configuration file where the user can define, modify or read partitions for all the operators he or she needs. Figure 5 shows a typical standard partition for the $\cap$ operator. As the reader may note, the user writes the standard partitions in a simple and intuitive LaTeX-based language. In this way Fastest implements one important feature of the TTF.

Another important aspect of Fastest is a key design decision that allows users to add new tactics. The only thing a user has to do is to program a class implementing the `Tactic` interface. This class has methods to configure and apply testing tactics.

```
\cap : operator(S,T);
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 \{\},
\lnot(S \subseteq T), \lnot (T \subseteq S), T \neq S;
end operator;
```

Fig. 5. Fastest's standard partition for $S \cap T$.

3.5 Fastest's Automatic Generation of Testing Trees

Once the engineer has entered the list of testing tactics for each operation, the only thing he or she has to do to build the testing tress is to run command `genalltt`. In other words, the sole manual work for the engineer, up to this point, is to enter the names of the tactics an other simple parameters: Fastest reads the definition of each testing tactic and builds automatically each node of each testing tree by manipulating the Z text.

3.6 Pruning Testing Trees in Fastest

Applying one tactic after another usually yield many empty test classes that is advisable to prune from the tree. Besides, the engineer might want to prune some nodes to reduce the number of test to run for various reasons. Then, a tool implementing the TTF should feature a way to automatically prune testing trees as well as some manual way. Fastest current version implements only manual pruning.

Automatic pruning involves some automatic way of finding contradictions. Although in general this is an unsolvable problem, we have envisioned a light-weight method –that is being implemented– that rests on the peculiarities of the problem within the TTF. We think that a full description of the method deserves its own paper, so here we just give some tips. First, we assume that test class predicates are conjunctions of atomic predicates. Second, Fastest would be delivered with a list of so called *elimination theorems*, which are parametrized conjunctions of atomic predicates that yield a contradiction –for instance $f = \varnothing \land \operatorname{dom} f \neq \varnothing$, where the name and type of f are parameters. Third, we let the user to extend the list with his own theorems. Fourth, Fastest applies all the elimination theorems for each test class, which means to seek the contradictory predicates inside the test class' predicate. Hence, if some class is a contradiction but Fastest fails to prune it, the user can add an elimination theorem that will prune similar test classes in future projects.

For manual pruning, Fastest provides three commands (`prunebelow`, `prunefrom` and `unprune`) that allow the engineer to prune subtrees and restore them in case of mistake. See Sect. 4 for more details.

3.7 Finding Abstract Test Cases with Finite Models

We have based our method for calculating abstract test cases from test objectives on the following:

Conjecture. For most of the predicates appearing in real specifications, a small finite model will suffice to find an element verifying that predicate; and if there is no element verifying the predicate, it is because the predicate is a contradiction.

Then, the method consists in building a finite model for each test class by calculating the Cartesian product between a very small finite set of values for each variable in the *VIS*. Fastest then evaluates –using ZLive– the class' predicate on each element of the model. Although this method might appear inefficient and is certainly inelegant, it has proved to be quite accurate as we show in Sect. 5.

In part the efficiency has been achieved by exploiting the highly parallelizable nature of the problem. In effect, Fastest asks each testing server to calculate a test case of a particular test class. Then, the time to find abstract test cases decrease proportionally with the number of available testing servers. This parallelization is so efficient because each calculation is completely independent from each other; synchronization is only needed when testing servers communicate a result to a client.

Then, every time Fastest asks a testing server to find an abstract test case from a given test class, it can happen one and only one of the following:

– Some element in the finite model satisfies the class' predicate, then we have the desired test case.
– There is no element in the finite model satisfying the predicate because the it is a contradiction –and was no pruned in the previous step.
– There is no element in the finite model satisfying the predicate, but the predicate is not a contradiction. In this case Fastest let the user to suggest a new finite model or part of it for the conflicting class (see command `setfinitemodel` in Sect. 4).

The key point here is what finite sets Fastest considers for each variable in the test class. By applying the first versions of the tool to some real case studies we were able to define a set of heuristics that make it possible to find a very high number of test cases, while minimizing the number of evaluations. These heuristics are the following:

– The finite sets for types $\mathbb{N}$ or $\mathbb{Z}$ are built from the numerical constants appearing in the predicate –if there are no such constants then we take $\{0, 1, 2\}$ and $\{-1, 0, 1\}$, respectively.
– The finite sets for enumerated types are the types themselves.
– The finite sets for basic types are built by generating three constant names of each type. For example, if $CHAR$ is a basic type, we assume the existence of $char1, char2, char3 : CHAR$, and then we take $\{char1, char2, char3\}$ as the finite set for any variable of type $CHAR$. We think of them as being three different constant elements.
– If a variable declared in the VIS does not appear in the predicate of a test class, then the finite set for that variable is any singleton –since the value of such a variable has no influence whatsoever on the evaluation of the predicate.
– If the predicate of a test class contains an atomic predicate of the form $var = VAL$ where var is a variable declared in the VIS and VAL is a constant value, then the finite set for var is just $\{VAL\}$ –since it will be impossible to satisfy the predicate with any other value.
– The finite sets for types such as $\operatorname{seq} X$, $X \pfun Y$, and so on, are built inductively from the finite sets considered for the more basic types.

The current version of Fastest does not cover the whole Z notation yet –in part because ZLives neither does. Hence, the Z models can only have variables of the following types: basic types, enumerated types, $\mathbb{N}$, $\mathbb{Z}$, set extensions, ranges, power sets, Cartesian products, binary relations, total functions, partial functions, sequences or any valid combination of them. For instance, Fastest does not yet support variables of schema types, $A \twoheadrightarrow B$, $A \rightarrowtail B$, $A \rightarrowtail\!\!\!\!\twoheadrightarrow B$, $\mathbb{P}_1\, A$ and $\mathbb{N}_1$. Axiomatic definitions are not supported neither. Although these are obvious limitations they do not impeded us to apply Fastest to real world examples as shown in Sect. 5.

4 Running an Example

In this section we use Fastest to run the example shown in Sect. 2. We will refer to some Z schema defined in that section.

Fastest is run from a command window with the following command:

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

Assuming the specification is stored in a file called `sensors.tex` it is loaded with:

```
loadspec sensors.tex
```

Before generating a testing tree the user needs to select one or more operations to test. In our example we select *KMR*.

```
selop KMR
```

Now it is time to apply the testing tactics. Fastest applies DNF by default. Standard Partitions is applied to the expression $smax\ s? < r?$ present in schema *KMROk* with:

```
addtactic KMR SP < smax~s? < r?
```

Test trees are generated with the command `genalltt`, which needs no parameters. Now the engineer can either prune the tree or calculate abstract test cases. Assume he or she wants to prune the tree. Since all the descendants of *KMR_DNF_2* contain an undefined expression $-smax\ s?$ because $s? \notin \operatorname{dom} smax-$ they can be pruned manually by issuing:

```
prunebelow KMR_DNF_2
```

while individual empty test classes can be pruned with `prunefrom`, as follows:

```
prunefrom KMR_SP_4
```

The testing tree resulting after pruning all the empty test classes is shown in Fig. 6, which was generated with command `showtt`.

Abstract test case generation is initiated by `genalltca`, without any parameters. After some time, the engineer can see that Fastest found an abstract test

```
KMR_VIS
  !______KMR_DNF_1
  |    !______KMR_SP_1
  |    !______KMR_SP_2
  |    !______KMR_SP_3
  |    !______KMR_SP_6
  |    !______KMR_SP_9
  |
  !______KMR_DNF_2
  |
  !______KMR_DNF_3
     !______KMR_SP_19
     !______KMR_SP_22
     !______KMR_SP_23
     !______KMR_SP_25
     !______KMR_SP_26
     !______KMR_SP_27
```

Fig. 6. Testing tree for *KMR* after pruning all the empty nodes.

case for all test objectives except for *KMR_SP_*1 and *KMR_SP_*9. The reason is that while Fastest will choose in this case the set $\{-1, 0, 1\}$ as the finite set for $\mathbb{Z}$, their predicates can only be satisfied by considering either two strictly positive or negative numbers. Hence, the user must help Fastest to find the right finite models. The commands are as follows:

```
setfinitemodel KMR_SP_1 -fm "\num == \{-1, -2\}"
setfinitemodel KMR_SP_9 -fm "\num == 1 \upto 2"
```

By running `genalltca` again Fastest will find abstract test cases for those two leafs.

As the reader can see, Fastest features an interesting level of automation. To date, the most human-time consuming task is pruning the testing tree, but automatic pruning will be included in the next version as we explained in Sect. 3.6.

5 Empirical Assessment

We do not assess MBT against other forms of testing, nor the relative suitability of the TTF because it has been done elsewhere [5, 16]. However, we think it is worth to mention that Fastest is the only one tool providing an implementation of the TTF were most of its steps are automatic, and that is one of the few tools for MBT for the Z notation [5] –see Sect. 6. Rather, what we want is to assess Fastest with respect to its ability to find abstract test cases from tests objectives. In particular we are interested in two measurements: (a) the percentage of non-empty test classes for which Fastest automatically finds an abstract test case;

and (b) the computing time necessary to calculate (a). Since testing trees are build almost instantaneously, we did not pay attention to this step.

Here we report the results of applying Fastest to two real-world, industrial-strength systems from the aerospace sector, and to five toy examples borrowed from the Z and MBT literature, and proposed by us as well. Tables 1 and 2 summarize the results of these experiments; note that both tables are related by column **N°**. In Sect. 5.1 we briefly describe each case study.

Table 1 gives an idea of the complexity of the models being "tested". **LOZC** stands for lines of Z code (in LaTeX format); **State** indicates the number of state variables in each model; and **Operations** gives the number of Z operations in each model. The last column shows the testing tactics that we have applied in the experiments. SISE is a tactic proposed by us that consist in partitioning an objective which contains a precondition of the form $v \in \{e_1, \dots, e_n\}$, where v is a variable and $e_1, \dots, e_n$ are expressions, in n new objectives each of which contains, also, a predicate of the form $v = e_i$. We applied at most two tactics to all operations in all experiments, except in one operation of the Scheduler case study, where three tactics were applied.

N°	**Case study**	**Real/Toy**	**LOZC**	**State**	**Operations**	**Tactics**
1	SWPDC	Real	1,238	18	17	DNF, SP, SISE, FT
2	Plavis	Real	608	13	13	DNF, SP, FT
3	Scheduler	Toy	240	3	10	DNF, SP
4	Security class	Toy	172	4	7	DNF, SP
5	Savings accounts	Toy	171	1	5	DNF, SP
6	Lift	Toy	152	6	3	DNF
7	Symbol table	Toy	78	1	3	DNF, SP

Table 1. Summary of the case studies (part one).

Table 2 gives an idea of the complexity of testing. **Classes** is the total number of test objectives right after applying the tactics, some of which were manually pruned, as is indicated in the third column, either because they are empty test classes or because they will not contribute significantly to test the target application. After pruning remains the **Possible** test objectives. Columns **Auto**, **Manual**, **Time** and **Ratio** are the most important for us. The first two of these indicates the number of test cases derived automatically and manually, respectively –by manually we mean that the engineer had to issue a `setfinitemodel` command to help Fastest to find a test case in a given test class. Percentages are with respect to column **Possible** because we want to assess the effectiveness of our method for finding abstract test cases from test classes. As the reader can observe, our conjecture (Sect. 3.7) was true even in the real-world experiments. In our opinion this implies some uniformity in specifications: although Z is a general purpose notation, users tend to specify using the same kind of predicates. Otherwise, a so simple heuristic could not have yield so high percentages.

It was quite surprising for us that this simple heuristic have been useful for specifications of systems of such diverse domains (see Sect. 5.1).

Before analyzing the last two columns of Table 2, it is necessary to give some details about the platform used for performing the experiments. All the experiments were conducted over the same hardware and software: an Intel Centrino Duo of 1.66 GHz with 1 Gb of main memory, running Linux Ubuntu 8.04 with kernel 2.6.24-24-generic and Java SE Runtime Environment (build 1.6.0_14-b08). Fastest was run in application mode –i.e. not in client-server mode–, what did not make use of parallelization, except for the concurrency that the Intel chip and the Linux kernel can provide. The last two columns show the total computing time (in minutes) necessary to find one test case for all **Possible** test objectives, and the computing time per test case, respectively. In our opinion, the amount of time is reasonable perhaps with the exception of experiment 2. The difference between both experiments may come from the fact that experiment 2 was the first real-world problem were Fastest was used. Now we know that by giving a few simple `setfinitemodel` commands, we could lower the time to the order of experiment 1. In fact, in experiment 1, we issued some `setfinitemodel` commands with the intention to reduce the time, and not because Fastest could not find the tests for itself.

N°	Classes	Pruned	Possible	Auto (%)	Manual (%)	Time	Ratio
1	225	123	112	91 (81.25%)	21 (18.75%)	158:00'	1:24'
2	232	128	117	104 (88.88%)	13 (11.11%)	1,111:00'	9:30'
3	213	174	29	28 (96.55%)	1 (3.44%)	3:00'	0:06'
4	36	16	20	20 (100%)	0 (0%)	0:20'	0:01'
5	97	74	23	23 (100%)	0 (0%)	4:50'	0:12'
6	17	1	16	16 (100%)	0 (0%)	7:30'	0:28'
7	18	10	10	10 (100%)	0 (0%)	0:10'	0:01'
Averages				**95.24%**	**4.76%**		**1:40'**

Table 2. Summary of the case studies (part two).

In spite of the good results we got so far, there is a key issue that we need to solve –in fact we are doing it as we write this article. The problem is how and how efficient can empty test classes be automatically pruned from testing trees. This point is important because, as the reader can see, there is a number of empty test classes in all the experiments. In all these case studies we analyzed one by one all the classes for which Fastest failed to find an abstract test case, just to find that close to 100% were actually empty –by the way, this fact supports the second part of our conjecture (Sect. 3.7). If most of the empty test classes cannot be pruned automatically, then Fastest either will require a non trivial amount of human effort, or the engineer could skip some test cases.

5.1 Description of the Case Studies

In this section we give a brief informal description of each case study.

SWPDC Is a simplified version of the communication protocol between two computers of a Brazilian satellite. The protocol roughly follows the directives of the ESA standard ECSS-E-70-41A. This model has operations to load a program in the memory of one of the computers, to transmit a data between the computers, to interact with some hardware devices, to dump the memory, and so on.

Plavis Is a simpler version of SWPDC.

Scheduler This model was borrowed from [5]. Basically, we translated the B model described in that book into Z.

Security class A security class is an computer security concept that belongs to mandatory access control [17]. A security class is composed of an element –called security level– that belongs to a totally ordered set and a set of elements –called categories– of other type. Then, there are operations to consult the security level or the set of categories, to change either one, etc.

Savings accounts This is a typical example of a Z model. A bank has many savings accounts where money can be deposited and withdrawn, the balance can be checked, a new account can be created and an existing account can be closed.

Lift This model was borrowed from [18].

Symbol table This model was borrowed from [2].

6 Related Work

In this section we compare Fastest with other similar tools. Clearly, TinMan [19, 20] is an obligated reference since is the implementation of the TTF by its own authors. However, TinMan is not as automatic as Fastest because: (a) the engineer has to enter by hand the predicates of the derived test classes, and (b) he or she has to enter by hand the values of the constants for the abstract test cases. In other words, as far as we understood, all the calculation is the engineer's responsibility while TinMan assist him or her in managing the specification, test objectives and abstract test cases. Although management of hundreds of test cases is a real problem, the automatic features of Fastest (a) avoids human error, and (b) saves human effort for the analytical work.

Confimiq launched in 2002 a tool called Conformiq Test Generator (CTQ) [21] which calculates and executes test cases from UML Statechars models. CQT was surpassed by a another tool by Conformiq named Qtronic. While CQT needed both a model of the environment of the system and a model of the system itself, Qtronic only needs a model of the system. Besides, Qtronic supports concurrent, multi-thread or non-deterministic models and time restrictions. Smartesting's Test Designer [22] is another tool that works with UML models. Test Designer also provides facilities for simulation. Two tools from IBM's Telelogic, Telelogic Statemate & Telelogic Rhapsody [23], also work from UML and

Statechart models. All these tools have many advantages with respect to Fastest, basically because they are the result of many years of development, but they have a non-trivial disadvantage. Statecharts are not particularly well suited to model data intensive systems and, strictly speaking, UML models are not formal, while Z has proved to be a very good formal notation for the specification of that kind of systems. In fact, every Statecharts model can be translated into a Z specification.

UniTESK Lab. is part of the Institute for System Programming of the Russian Academy of Sciences. This laboratory has developed a couple of similar MBT tools named CTESK and JavaTESK. These tools use models written in extensions of the C and Java programming languages developed also at UniTESK. As far as we understand both tools are aimed at system testing and not particularly for unit testing. ModelJUnit is an extension of JUnit that supports MBT based on FSM written in Java [24]. An obvious limitation of these tools is their dependency on programming languages, but it also can be their strength since programmers might find it easier to write models in languages similar to the ones they routinely work with.

Rave [25] uses tabular models derived from the work done by David L. Parnas at the U.S Naval Research Laboratory. T-VEC, the company which developed Rave, also advertises T-VEC Tester [26] which uses Simulink and Stateflow models. Reactis [27], from Reactive Systems, also uses Simulink and Stateflow models from which it generates test cases.

In the open source community we found a Java implementation of MBT called org.tigrismbt. This tool generates sequences of tests from FSM written as graphs in graphml format.

LTG/B from LEIRIOS (now Smartesting) [5] shares some similarities with Fastest. This tool uses B models –which are similar to Z models– [28] and is based on the symbolic animation of formal specifications and two test generation strategies: analysis of cause-effect and analysis of boundaries. One difference with Fastest is that LTG/B generates sequences of operations that put the system in the desired state, rather than giving the values of state variables to put the system in that state, as the TTF suggests.

For an up-to-date list of MBT tools consult [29].

As this survey shows there are no implementations of MBT tools for the Z formal notation as automatic as Fastest. This situation combined with the relative widespread of the Z formal notation in industry and the advantages of the TTF, led us to implement the framework to provide such a tool to the Z community.

7 Conclusions and Future Work

We presented a tool, called Fastest, that automates the generation of unit tests from Z models based on the Stocks-Carrington framework for model-based testing. A simple method is proposed to find abstract test cases from test objectives and an empirical analysis based on seven models from different domains is also shown. The architecture of the tool was briefly introduced and a toy example was executed. We think that the architecture presents some interesting features such as the possibility of distributing the calculation of test cases, the way by means of which new testing tactics can be added and the existing ones can be configured, the integration with the CZT project, etc. It is worth to mention that Fastest is still a prototype that needs to be extended in many ways.

As the empirical study shows, we can conclude that, an apparently rough method to calculate abstract test cases, based on a practical conjecture about the form of real-world specifications, proved to be very accurate and reasonable efficient and, combined with the architecture, scalable as well. On the other side, the direct implementation of the TTF framework made it possible to take advantage of all its interesting properties.

However, among others, we need to improve on automatic pruning of test trees to dramatically reduce the amount of human effort during the testing process. We also sketched our ideas on this regard. Also, more testing tactics proposed within the TTF, such as specification mutation and sub-domain propagation, have to be implemented. The user interface prays for an improvement. Finally, it will be necessary to support the whole Z notation, but this is tied, to some extent, to the improvement of CZT's ZLive animator.

References

1. Rodriguez Monetti, P., Cristiá, M.: Fastest. http://www.flowgate.net/?lang=en&seccion=herramientas
2. Stocks, P., Carrington, D.: A framework for specification-based testing. IEEE Transactions on Software Engineering **22**(11) (November 1996) 777–793
3. Stocks, P.: Applying formal methods to software testing. PhD thesis, Department of Computer Science, University of Queensland (1993)
4. Maccoll, I., Carrington, D.: Extending the test template framework. In: Proceedings of the third northern formal methods workshop. (1998)
5. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2006)
6. Spivey, J.M.: The Z notation: a reference manual. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1989)
7. Souza, S., Maldonado, J., Fabbri, S., Masiero, P.: Statecharts specifications: A family of coverage testing criteria. In: CLEI'2000 - XXVI Latin-American Conference of Informatics, CLEI (2000)
8. Bernard, E., Legeard, B., Luck, X., Peureux, F.: Generation of test sequences from formal specifications: GSM 11-11 standard case study. International Journal of Software Practice and Experience **34**(10) (2004) 915–948
9. Dick, J., Faivre, A.: Automating the generation and sequencing of test cases from model-based specifications. In: FME '93: Proceedings of the First International Symposium of Formal Methods Europe on Industrial-Strength Formal Methods, London, UK, Springer-Verlag (1993) 268–284
10. Malik, P., Utting, M.: CZT: A framework for Z tools. In: ZB. Lecture, Springer (2005) 65–84
11. Community Z Tools. http://http://czt.sourceforge.net

12. ISO: Information technology – z formal specification notation – syntax, type system and semantics. Technical Report ISO/IEC 13568, International Organization for Standardization (2002)
13. Clements, P., Garlan, D., Bass, L., Stafford, J., Nord, R., Ivers, J., Little, R.: Documenting Software Architectures: Views and Beyond. Pearson Education (2002)
14. Shaw, M., Garlan, D.: Software architecture: perspectives on an emerging discipline. Prentice Hall, Upper Saddle River (1996)
15. Garlan, D., Kaiser, G.E., Notkin, D.: Using tool abstraction to compose systems. Computer **25**(6) (1992) 30–38
16. Legeard, B., Peureux, F., Utting, M.: 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, London, UK, Springer-Verlag (2002) 309–329
17. Bishop, M.: Computer Security. Art and Science. Addison Wesley (2003)
18. Hörcher, H.M., Peleska, J.: Using formal specifications to support software testing. Software Quality Journal **4** (1995) 309–327
19. Murray, L., Carrington, D., Maccoll, I., Strooper, P.: Tinman - a test derivation and management tool for specification-based class testing. In: In Technology of ObjectOriented Languages and Systems (TOOLS. (1999) 222–233
20. Murray, L.: Software requirements specification for tinman - version 1.0. Technical Report 99-02, The Univesity of Qeensland (1999)
21. Conformiq. http://www.conformiq.org
22. Smartesting. http://www.smartesting.com
23. Telelogic: Telelogic statemate & telelogic rhapsody. http://modeling.telelogic.com/products/
24. Utting, M.: Model junit. http://www.cs.waikato.ac.nz/ marku/mbt/modeljunit/
25. T-VEC: Requirements-based automated verification. http://www.t-vec.com/solutions/rave.php
26. T-VEC: T-vec tester for simulink and stateflow. http://www.t-vec.com/solutions/simulink.php
27. Reactis. http://www.reactive-systems.com
28. Abrial, J.R.: The B-book: assigning programs to meanings. Cambridge University Press, New York, NY, USA (1996)
29. Utting, M.: Commercial MBT tools. www.cs.waikato.ac.nz/ marku/mbt/CommercialMbtTools.pdf