SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2010; **00**:1–34
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr

Tool Support for the Test Template Framework

Maximiliano Cristiá[1]*, Pablo Albertengo[2] , Claudia Frydman[3] ,
Brian Plüss[4] and Pablo Rodríguez Monetti[2]

[1]*CIFASIS and UNR, Argentina*
[2]*FCEIA-UNR, Argentina*
[3]*CIFASIS-LSIS and AMU, France*
[4]*Centre for Research in Computing, The Open University, UK*

SUMMARY

This paper describes tool support that has been implemented for the Test Template Framework (TTF). The TTF is a model-based testing (MBT) method specially well-suited for unit testing from Z specifications. Although the TTF is a sound MBT method and it has been widely referenced since its first publication, attention in recent years has decayed. In fact, some have argued that generating abstract test cases following the TTF is a manual task requiring that its users perform complex predicate manipulations. This paper shows that these observations are dubious by describing Fastest, a tool that implements solutions for all these issues and, according to many experiments, produces abstract test cases for more than 80% of the satisfiable test specifications. Furthermore, it is claimed that Fastest fulfils the needs of the Z user community regarding MBT tools, which is supported with a range of case studies. Copyright © 2010 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: model-based testing; test template framework; Z notation; Fastest; tool support

1. INTRODUCTION

Model-based testing (MBT) is a well-known technique aimed at testing software systems analysing a formal model [1, 2]. MBT approaches start with a formal model or specification of the software, from which test cases are generated, as shown in Fig. 1. These techniques have been developed and applied to models written in different formal notations such as Z [3], finite state machines and their extensions [4], B [5], algebraic specifications [6], and so on. 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. The rest of the paper assumes that the audience is familiar with MBT, otherwise the reader may consult Utting and Legeard [1].

Phil Stocks and David Carrington [3, 7, 8] introduced a formal framework to conduct MBT from Z specifications [9] called Test Template Framework (TTF). The TTF is a particular method for the "Generation" step of the MBT process (Figure 1), specially well-suited for unit testing. As is noted by Legeard et al. [5], the TTF method was 'quite widely known and referenced since' its first international publication. However, it may be argued that the lack of tool support made the MBT community lose interest in it. In effect, Legeard and his colleagues conclude, after applying the TTF to a case study, that 'TTF generation is a manual process, requiring extensive expertise at manipulating and simplifying schemas'.

*Correspondence to: CIFASIS, Bv. 27 de febrero 210 bis, Rosario, Argentina. E-mail: cristia@cifasis-conicet.gov.ar

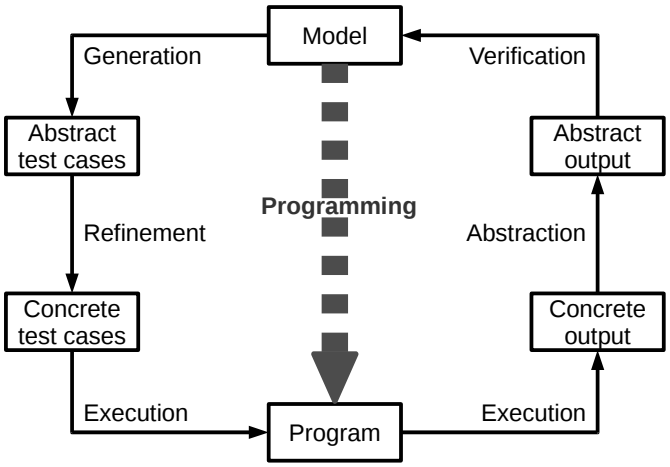


Figure 1. A general description of the MBT process.

On the other hand, Z users find the TTF appealing and sound. Particularly, they find a great advantage in the TTF since all the MBT main artefacts—test specifications, abstract test cases, etc.—can be expressed in the same notation of the model, i.e. the Z formal notation. After studying the TTF, the authors concluded that it might be automated roughly to the same level of any other MBT method. Furthermore, providing tool support for it would contribute to fill two gaps: (a) MBT tools approaching unit testing, and (b) a MBT tool for the Z user community.

The result of this work is a tool, called Fastest, that implements the TTF, allowing users to semi-automatically produce abstract test cases from Z specifications, which, in turn, enables the semi-automation of other tasks, like translating test cases into natural language. In this paper, an integral presentation of previous work [10, 11, 12] is made and new results are introduced as well.

According to the TTF: (i) test specifications are assembled into so-called *testing trees*, generated by applying *testing tactics*; (ii) testing trees should be pruned to avoid unsatisfiable test specifications; and (iii) abstract test cases are derived from the remaining leaves of these testing trees. These three main features of the TTF required the implementation of complex predicate manipulation functions dealing with Z constructs such as schema boxes and types [5].

Obviously, test cases generated by Fastest are paragraphs of formal text. While this description is suitable for the automatic tasks involved in testing, it requires that stakeholders be able to read Z specifications in order to understand what is being tested. Hence, since it contributes to provide tool support for the TTF, this paper includes a description of a natural language generation (NLG) [13] template-based method to automatically translate abstract test cases into natural language [12].

The main contribution of the paper is the introduction of a tool supporting key tasks of the TTF which, as far as it has been investigated, did not exist before. More precisely, the tool gives support for the following tasks: (a) testing tactics can be applied automatically; (b) inconsistent test specifications can be semi-automatically eliminated from testing trees; (c) abstract test case generation can be performed semi-automatically; and (d) abstract test cases can be semi-automatically translated into natural languages. It is important to remark that Stocks and Carrington did not propose how to automate (b) and (c)—and they never mentioned (d). The results of running Fastest on eleven experiments, two of which are industrial-strength case studies, are also shown.

The paper is structured as follows. The next section presents an introduction to the TTF by means of an example. In Section 3 the functional and architectural features of Fastest are introduced. Sections 4, 5 and 6 describe, respectively, the three main features of the TTF requiring predicate manipulation. The NLG method proposed to translate test cases into natural language is introduced in Section 7. Then, in Section 8, the degree of automation and limitations of the tool are discussed. Section 9 presents the results of applying Fastest to eleven case studies. This work is compared to similar approaches in Section 10. Finally, Section 11 describes the conclusions and future work.

DOI: 10.1002/stvr

2. THE TEST TEMPLATE FRAMEWORK

This section introduces the TTF by means of an example that will be used throughout this article. The TTF is described without considering any particular implementation or tool. It is assumed that the reader is fluent in the Z notation. As it has been said, the TTF is a particular method for the "Generation" step of the MBT process (Figure 1), specially well-suited for unit testing from Z specifications. Each operation within the specification is analysed to derive or generate abstract test cases. This analysis consists of the following steps (that will be detailed in the next subsections):

1. Consider the valid input space (VIS) of each Z operation.
2. Apply one or more testing tactics in order to partition the input space.
3. Build a tree of test specifications.
4. Prune inconsistent test specifications.
5. Find one abstract test case from each remaining test specification.

One of the main advantages of the TTF is that all of these concepts are expressed in the same notation of the specification, as it will be shown shortly. Hence, the engineer has to know only one notation to perform the analysis down to the generation of abstract test cases.

2.1. A Z Specification of a Simple Savings Accounts System

Think about 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. The bank only needs to keep record of the balance of each account, and the ID and name of each client. Any person can open an account at the bank, becoming the account's first owner. Account owners can add and remove other owners to their accounts, withdraw money and check the balance. Any person can deposit money in any accounts. To keep the example manageable, only one of these operations will be formalized.

2.1.1. Basic Types. As expressed in the previous section, each savings account is identified by an account number. Since account numbers are used just as identifiers, they can be abstracted away, ignoring their internal structure. Therefore, account numbers are introduced as a basic type:

$$[ACCNUM]$$

Along the same lines, basic types for the ID's of clients and their names are also introduced:

$$[UID, NAME]$$

The money that clients can deposit and withdraw, and the balance of savings accounts are represented as natural numbers. 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. If the decimal positions are really needed, then think that each natural number used in the specification is the result of multiplying the corresponding real number by a convenient power of 10—for instance, 100. Then:

$$MONEY == \mathbb{N}$$
$$BALANCE == \mathbb{N}$$

In other words, two synonyms are introduced for the set of natural numbers to make the specification more readable. Finally, an enumeration of different outputs that operations will perform in order to communicate whether they were successful or not is defined.

$$MSG ::= ok \mid clientAlreadyExists \mid accountAlreadyExists$$

The meaning of these constants will become clear when the operation is formalized.

DOI: 10.1002/stvr

2.1.2. The State Space. The state space is defined as follows:

$$
\begin{array}{|l}
\hline
Bank \\
clients : UID \pfun NAME \\
balances : ACCNUM \pfun BALANCE \\
owners : UID \rel ACCNUM \\
\hline
\end{array}
$$

It makes sense to define *clients* as a function because each person is uniquely identified by his/her *UID*, but two different persons might have the same name; and it make sense to make *clients* partial because not every person is a client of the bank at all times. The same is valid for *balances*: there is a functional relationship between account numbers and balances, and not all the account numbers are used at all times at the bank. On the other hand, 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 several clients.

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

$$
\begin{array}{|l}
\hline
InitBank \\
Bank \\
\hline
clients = \varnothing \\
balances = \varnothing \\
owners = \varnothing \\
\hline
\end{array}
$$

2.1.3. State Invariants. A predicate is said to be a state invariant if it holds in every state of the system. The usual Z style includes state invariants in the state schema. For example, the state schema for the savings account system would have been:

$$
\begin{array}{|l}
\hline
Bank \\
clients : UID \pfun NAME \\
balances : ACCNUM \pfun BALANCE \\
owners : UID \rel ACCNUM \\
\hline
\mathrm{dom}\, clients = \mathrm{dom}\, owners \\
\mathrm{dom}\, balances = \mathrm{ran}\, owners \\
\hline
\end{array}
$$

instead of the one defined earlier. However, for reasons that are going to be explained in Section 2.7, in this paper state invariants are treated in a different fashion, as proposed, for instance, in the Z/EVES system [14, page 27]. First, a schema stating the invariant is defined:

$$
\begin{array}{|l}
\hline
BankInv \\
Bank \\
\hline
\mathrm{dom}\, clients = \mathrm{dom}\, owners \\
\mathrm{dom}\, balances = \mathrm{ran}\, owners \\
\hline
\end{array}
$$

and then, a proof obligation stating that each operation preserves the invariant is required. For example:

Theorem NewClientInv

$$(InitBank \Rightarrow BankInv) \land (BankInv \land NewClient \Rightarrow BankInv')$$

Discharging such proof obligations is a responsibility of those who write the specification. This way of writing invariants is similar to other formal methods, such as TLA+ [15] and B [16].

DOI: 10.1002/stvr

2.1.4. Opening the First Savings Account. Now it is possible to start defining the operations of the system. However, as it has been said earlier, only the specification of the operation that describes how a person can open his/her first savings account will be introduced. The first schema represents the person fulfilling all the conditions to open his/her first account:

NewClientOk

$$\Delta Bank$$
$$u? : UID$$
$$name? : NAME$$
$$n? : ACCNUM$$
$$msg! : MSG$$

$$u? \notin \mathrm{dom}\, clients$$
$$n? \notin \mathrm{dom}\, balances$$
$$clients' = clients \cup \{u? \mapsto name?\}$$
$$balances' = balances \cup \{n? \mapsto 0\}$$
$$owners' = owners \cup \{u? \mapsto n?\}$$
$$msg! = ok$$

Input variable $u?$ represents the ID of the person willing to open a savings account at the bank, and $name?$ is his/her name. To simplify the specification it is assumed that a bank clerk provides the account number $n?$ when the operation is called—instead of the system generating it. As it can be seen, the bank requires that the person willing to open the account is not a client and that the account number is not in use.

However, *NewClientOK* does not say what the system should do when $u? \notin \mathrm{dom}\, clients \land n? \notin \mathrm{dom}\, balances$ does not hold. The bank says that nothing has to be done when either the person requesting the account is already a client or when the account number chosen by the clerk is already in use. Then, a new schema for the first case is introduced:

$$ClientAlreadyExists == [\Xi Bank;\ u? : UID \mid u? \in \mathrm{dom}\, clients \land msg! = clientAlreadyExists]$$

and the following schema for the negation of the remaining precondition:

$$AccountAlreadyExists == [\Xi Bank;\ n? : ACCNUM \mid n? \in \mathrm{dom}\, balances \land msg! = accountAlreadyExists]$$

Finally, these three schemas are assembled together to define the total operation—i.e. an operation whose precondition is equivalent to *true*—for a person opening his/her first savings account at the bank:

$$NewClient == NewClientOk \lor ClientAlreadyExists \lor AccountAlreadyExists$$

2.2. The Valid Input Space of a Z Operation

The Valid Input Space (VIS) of a Z operation is derived from its Input Space (IS). The IS is defined to be the Z schema whose declaration includes all the input and (non-primed) state variables declared in the operation and no predicate. The IS for *NewClient* is the following schema:

$$NewClient_{IS} == [clients : UID \pfun NAME;\ balances : ACCNUM \pfun BALANCE;\ owners : UID \rel ACCNUM;\ u? : UID;\ name? : NAME;\ n? : ACCNUM]$$

Prepared using stvrauth.cls
DOI: 10.1002/stvr

$$
\begin{array}{ll}
S = \varnothing, T = \varnothing & S \neq \varnothing, T \neq \varnothing, S \subset T \\
S = \varnothing, T \neq \varnothing & S \neq \varnothing, T \neq \varnothing, T \subset S \\
S \neq \varnothing, T = \varnothing & S \neq \varnothing, T \neq \varnothing, T = S \\
S \neq \varnothing, T \neq \varnothing, S \cap T = \varnothing & S \neq \varnothing, T \neq \varnothing, S \cap T \neq \varnothing, \neg (S \subseteq T), \\
 & \quad \neg (T \subseteq S), S \neq T
\end{array}
$$

Figure 2. Standard partition for $S \cup T$, $S \cap T$ and $S \setminus T$.

The VIS of an operation is the schema that restricts the IS to verify the precondition of the operation:

$$Op_{VIS} == [Op_{IS} \mid \text{pre } Op]$$

Given that the precondition of *NewClient* is equivalent to *true*, the VIS is equal to the IS:

$$NewClient_{VIS} == NewClient_{IS}$$

2.3. Applying Testing Tactics

Stocks and Carrington [3] suggest dividing the VIS into equivalence classes by applying one or more *testing tactics*. These equivalence classes are called *test classes*, *test objectives*, *test templates* or *test specifications* in the literature. The latter will be used in this paper. Test specifications obtained in this way can be further subdivided into more test specifications by applying other testing tactics. The net effect of this technique is a progressive partition of the VIS into more restrictive test specifications. This procedure can continue until the engineer is satisfied with the possible accuracy of the test specifications with respect to their ability to uncover errors in the implementation.

Therefore, testing tactics are the tools that engineers use to partition the VIS of each operation. A tactic indicates how the current test specification should be subdivided, by giving the set of predicates that defines each partition. Two of the testing tactics proposed within the TTF are the following (see more about these and other testing tactics in Section 4):

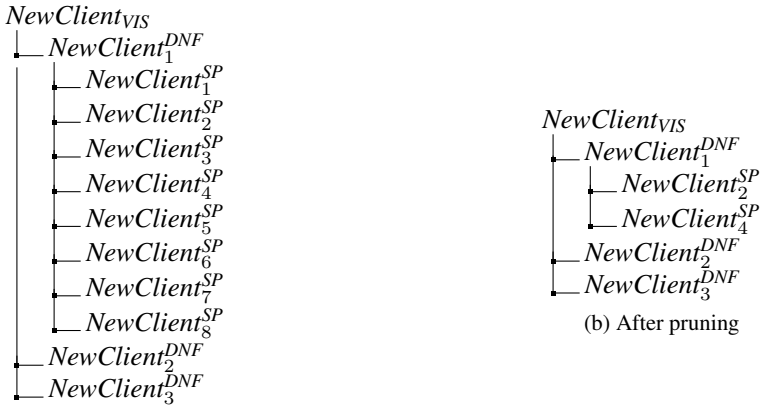
- Disjunctive Normal Form (DNF). To apply this tactic, first write the predicate of the operation in DNF. The set of predicates that characterises this tactic is the precondition of each disjunct in the resulting DNF.
- Standard Partitions (SP). This tactic is parametrised by a mathematical operator. Then, the set of predicates indicated by the tactic to partition the test specification depends on the operator. Figure 2 shows the predicates proposed by the standard partition of $\cup$, $\cap$ and $\setminus$.

For example, applying DNF to the VIS of NewClient, generates the following test specifications (note that test specifications are also written in Z):

$$
\begin{aligned}
NewClient_1^{DNF} &== [NewClient_{VIS} \mid u? \notin \operatorname{dom} clients \land n? \notin \operatorname{dom} balances] \\
NewClient_2^{DNF} &== [NewClient_{VIS} \mid u? \in \operatorname{dom} clients] \\
NewClient_3^{DNF} &== [NewClient_{VIS} \mid n? \in \operatorname{dom} balances]
\end{aligned}
$$

Now, SP is applied to $\cup$ in $clients \cup \{u? \mapsto name?\}$ in order to partition $NewClient_1^{DNF}$, yielding the following test specifications (note how test specifications are linked by schema inclusion):

DOI: 10.1002/stvr



(a) Initial

(b) After pruning

Figure 3. Initial and pruned testing trees of *NewClient*.

$$NewClient_1^{SP} == [NewClient_1^{DNF} \mid clients = \varnothing \land \{u? \mapsto name?\} = \varnothing]$$
$$NewClient_2^{SP} == [NewClient_1^{DNF} \mid clients = \varnothing \land \{u? \mapsto name?\} \neq \varnothing]$$
$$NewClient_3^{SP} == [NewClient_1^{DNF} \mid clients \neq \varnothing \land \{u? \mapsto name?\} = \varnothing]$$
$$NewClient_4^{SP} == [NewClient_1^{DNF} \mid clients \neq \varnothing \land \{u? \mapsto name?\} \neq \varnothing \land clients \cap \{u? \mapsto name?\} = \varnothing]$$
$$NewClient_5^{SP} == [NewClient_1^{DNF} \mid clients \neq \varnothing \land \{u? \mapsto name?\} \neq \varnothing \land clients \subset \{u? \mapsto name?\}]$$
$$NewClient_6^{SP} == [NewClient_1^{DNF} \mid clients \neq \varnothing \land \{u? \mapsto name?\} \neq \varnothing \land \{u? \mapsto name?\} \subset clients]$$
$$NewClient_7^{SP} == [NewClient_1^{DNF} \mid clients \neq \varnothing \land \{u? \mapsto name?\} \neq \varnothing \land clients = \{u? \mapsto name?\}]$$
$$NewClient_8^{SP} == [NewClient_1^{DNF} \mid clients \neq \varnothing \land \{u? \mapsto name?\} \neq \varnothing \land clients \cap \{u? \mapsto name?\} \neq \varnothing \land \neg (clients \subseteq \{u? \mapsto name?\}) \land \neg (\{u? \mapsto name?\} \subseteq clients)]$$

2.4. *Building a Tree of Test Specifications*

According to the TTF, the test specifications of a given operation must be organized in *testing trees*. A testing tree has the VIS at its root, the test specifications resulting from applying the first tactic at the next level, and so forth. The testing tree of *NewClient* is shown in Figure 3a.

Testing trees are important because the TTF prescribes deriving abstract test cases only from their leaves. This is so because each leaf conjoins the predicate of the test specifications above it up to the root, thus making it unreasonable to derive abstract test cases from the internal nodes. For instance, if $NewClient_1^{DNF}$ is unfolded in $NewClient_4^{SP}$, the result is as follows:

$$NewClient_4^{SP} == [NewClient_{VIS} \mid u? \notin \operatorname{dom} clients \land n? \notin \operatorname{dom} balances \land clients \neq \varnothing \land \{u? \mapsto name?\} \neq \varnothing \land clients \cap \{u? \mapsto name?\} = \varnothing]$$

These trees can be automatically obtained from the test specifications, as children include a reference to their parent node by schema inclusion.

DOI: 10.1002/stvr

2.5. Prune Inconsistent Test Specifications

Some test specifications might be empty because their predicates are unsatisfiable. In these cases, it is impossible to find abstract test cases. Inconsistent test specifications must be pruned from the tree. For instance, $NewClient_1^{SP}$ is inconsistent because $\{u? \mapsto name?\}$ cannot be empty. Another example is $NewClient_7^{SP}$, because *clients* cannot be equal to $\{u? \mapsto name?\}$ since $u? \notin \mathrm{dom}\, clients$ also holds. In this example, the testing tree resulting after pruning is depicted in Figure 3b.

2.6. Derive Abstract Test Cases from Test Specifications

Finally, the engineer has to choose at least one element satisfying each of the remaining leaves of the testing tree. These are the abstract test cases. For example, the following horizontal schemas represent (some of) the abstract test cases of the corresponding test specifications. Identifiers such as *name*0 or *uid*0 are assumed to be declared in axiomatic definitions. They are considered as constants of their types, and are assumed to be different if they have different names.

$$
\begin{aligned}
&NewClient_1^{TC} == \\
&\quad [NewClient_2^{SP} \mid \\
&\qquad u? = uid0 \land n? = accnum0 \land name? = name0 \\
&\qquad \land\ owners = \varnothing \land balances = \varnothing \land clients = \varnothing] \\
&NewClient_2^{TC} == \\
&\quad [NewClient_4^{SP} \mid \\
&\qquad u? = uid0 \land n? = accnum0 \land name? = name0 \\
&\qquad \land\ owners = \varnothing \land balances = \varnothing \land clients = \{(uid1, name0)\}] \\
&NewClient_3^{TC} == \\
&\quad [NewClient_2^{DNF} \mid \\
&\qquad u? = uid0 \land n? = accnum0 \land name? = name0 \\
&\qquad \land\ owners = \varnothing \land balances = \varnothing \land clients = \{(uid0, name0)\}] \\
&NewClient_4^{TC} == \\
&\quad [NewClient_3^{DNF} \mid \\
&\qquad u? = uid0 \land n? = accnum0 \land name? = name0 \\
&\qquad \land\ owners = \varnothing \land balances = \{(accnum0, 0)\} \land clients = \varnothing]
\end{aligned}
$$

Once again, note that abstract test cases are also written in Z, and that they are linked to test specifications by schema inclusion.

2.7. Brief Discussion of the TTF

The TTF is particularly appealing for Z users since it keeps all the key elements (operations, test specifications, abstract test cases and others) within the Z notation. Also it naturally provides traceability between all these elements by using schema inclusion. Besides, users can define new testing tactics that best fit their needs when the standard ones fall short.

As it can be seen, within the TTF an abstract test case is a conjunction of equalities between VIS variables and constant values, rather than a sequence of operations leading to the desired state, as it is suggested by other approaches [17, 18, 19]. These sequences are useful when the system under test (SUT) has to be put in a particular state so that a test can be run from it. Since the TTF does not produce such sequences, an alternative is to proceed as follows: (a) refine a TTF abstract test case into an executable program; and (b) set the initial state of the SUT according to the abstract test case. Cristiá and colleagues [20] have proposed and implemented such a method for the Fastest tool, whose only prerequisite is the availability of the SUT's source code.

As it has been shown in Section 2.1.3, state invariants are written in a separate schema and not in the predicate part of the state schema. In this way, when the TTF is applied to an operation, state invariants are not considered. In other words, state invariants are not analysed in the process of generating test cases. This seems to imply that the code implementing such invariants will not be tested. However, as noted by [21, page 6], invariants are not part of a program implementing

DOI: 10.1002/stvr

the specification, so it is not strictly necessary to analyse them in order to generate test cases. In the approach proposed here for recording state invariants, for each operation there should be a proof guaranteeing that the operation satisfies the state invariant. This means that each operation must have been specified in such way that it verifies the state invariant. So, when the operation is analysed following the TTF, the test cases generated will test code implementing functionality that is sufficient to make the program verify the state invariant—because the program implements its specification and the specification satisfies the invariant. More formally, if operation O satisfies invariant I and it is "shown" by testing that program P implements O, then it can be said that P satisfies I. In summary, there is no need to consider state invariants in the "Generation" step, as long as the corresponding proof obligations have been discharged.

Both the TTF and Fastest work well with either form of writing state invariants. Nevertheless, they both work better if the proposed approach is followed. This is because, although including state invariants in the state schemas makes operations more complex, this complexity does not result in better testing, as it has been explained in the previous paragraphs.

3. FASTEST: TOOL SUPPORT FOR THE TTF

The main contribution of this article is to show that it is possible to provide tool support for the TTF, thus making many of its steps semi-automatic. The result of this work is a tool, called Fastest, which gives support to all of the TTF steps. The goal of this section is to give an overview of Fastest by describing its functional and architectural features, as well as the technology that has been used. Latter sections provide more detailed descriptions of the main features.

Fastest is open-source software and is available online [22]. The tool does not cover the Z notation in full, although it supports a very general and expressive subset [23, Appendix A].

3.1. Conceptual Description

Fastest receives a Z specification and lets users issue commands to produce abstract test cases for selected operations. The specification must be written in the standard Z LaTeX mark-up language [9]. Specifications can be written using any text editor. The authors have chosen Eclipse [24], with the CZT [25] and TeXlipse [26] extensions, as it provides seamless integration of Latex and Z syntax checking, as well as type checking for Z, together with many other IDE functionalities. Once the specification is successfully loaded into Fastest, it is transformed into an internal representation more suitable for parsing and static analysis. This transformation is performed using tools from the CZT project [27, 28], as it will be shown shortly. Fastest includes commands for exporting all of the artefacts produced during a session to LaTeX mark-up.

Another key conceptual aspect of Fastest is extensibility. Users are allowed to extend the tool in several ways. The pruning method, for instance, can be easily improved to make it more powerful as more projects are carried on. Users can define new standard partitions or modify the existing ones by simply editing a LaTeX file. Furthermore, if they want to add new testing tactics they can implement a Java interface and plug in the resulting class into the tool. All of these properties will be detailed in coming sections.

The tool also includes functionality to perform the "Refinement", "Execution" and "Abstraction" steps of Figure 1, but their description is beyond of the scope of this paper.

3.2. System Architecture

Fastest was envisioned as a client-server Java application [29]. The main reason for thinking of a distributed system came from the realisation that calculating abstract test cases from test specifications in large projects can be a hard computing problem but also highly parallelisable. For this reason, an scalable application using the idle computer power present in a corporate network, is proposed. Moreover, since testing a large application is usually carried on by a group of testers, a client-server architecture allows for the team to work concurrently from different workstations by running one client process per engineer. However, in such large projects there is

Prepared using stvrauth.cls
DOI: 10.1002/stvr

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 may have a data server that is known to all other processes, some client processes and a number of testing servers.

The client is organized by following the Implicit Invocation architectural style [29]. 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 of adding, removing and changing a component without affecting others [29]. For instance, it would be quite easy to add a component, called whenever a new test case is calculated, that stores it on disc or that refines it into a given programming language. Users interact with the application through the clients. The user interface of the client software is text-based, similar to command-line applications like Linux's bash, from which users can issue commands.

3.3. The CZT Framework

The Community Z Tools (CZT) project [27, 28] is an open-source Java framework created in 2003 with the goal of building a toolkit for the Z notation and 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 Syntax Trees (AST)
- CZT Type-checker
- CZT ZLive specification animator

Although Fastest benefited greatly from all these services, perhaps the most important was ZLive. The ZLive tool 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. This service is used to calculate abstract test cases as described in Section 6.

4. BUILDING TESTING TREES WITH TESTING TACTICS

As it has been earlier, test specifications are assembled into testing trees by applying testing tactics. Fastest features two user commands, `addtactic` and `genalltt`, that allow users to build testing trees. `addtactic` receives the name of a Z operation or the name of an existing test specification, a tactic name and its parameters. `genalltt` applies the tactic producing all the test specifications indicated by it. Users can either run `addtactic` many times thus queuing tactics to be applied later, or they can run `addtactic` and `genalltt` iteratively to produce the same result. In other words, the sole manual work for the engineer, up to this point, is to enter the names of the tactics and other simple parameters.

Currently, Fastest implements DNF, Standard Partitions and some new tactics proposed during the course of this research. Next sections will describe how Fastest calculates the DNF of an operation, how users can define their own standard partitions, the rationale behind the introduction of the new testing tactics and how engineers wishing to extend the tool can add their own testing tactics.

4.1. Disjunctive Normal Form

Writing a predicate in DNF means writing it as a disjunction of conjunctions of atomic predicates or negations of atomic predicates—quantified predicates are considered atomic. There is a very well-known algorithm for writing a predicate in DNF [30]. However, writing a predicate in DNF may lead to an exponential problem. For example, the DNF of predicates of the form:

$$(P_1 \lor Q_1) \land (P_2 \lor Q_2) \land \ldots \land (P_n \lor Q_n) \tag{1}$$

have 2^n terms. Some terms may be simplified by applying simple logical rules but in general the exponential problem remains; unless engineers simplify the predicate by using a proof assistant.

DOI: 10.1002/stvr

Since the intention is to keep DNF calculation fully automatic Fastest implements an algorithm that ameliorates the exponential problem by giving, in some cases, an incomplete DNF.

In Z specifications schema inclusion is heavily used to structure the model; sometimes several levels of schema inclusion can be found. In large specifications top-level operations are defined by schema expressions referencing other schemas that in turn point to more schemas, and so forth. None of these schemas have a complex explicit predicate, but when all the schemas referenced in one of these top-level schemas are unfolded the resulting predicate can be quite large and complex. All this sums up to the fact that when the DNF of a complex Z operation is calculated, there are great chances of having an exponential number of terms in it. Therefore, Fastest implements some heuristics to avoid it as much as possible.

The first heuristic is aimed at not unfolding all the schemas. If users select schema A as an operation to be tested and it appears in the definition of another schema B, then when the DNF of B is calculated A is considered as an atomic predicate, and in consequence is not unfolded—operation selection is explained in Section 8. The rationale behind this heuristic is two fold: (a) if A was selected to be tested then test cases for it will be generated regardless of unfolding it in B or not; and (b) not unfolding A in B keeps B's predicate more simple contributing to reduce the size of its DNF.

Before considering the second heuristic note that the DNF of (1) can be calculated iteratively by distributing $\vee$ over $\wedge$:

$$\begin{aligned}&(P_1 \wedge P_2 \wedge \ldots \wedge P_n)\\ &\vee (P_1 \wedge P_2 \wedge \ldots \wedge P_{n-1} \wedge Q_n)\\ &\vee \ldots\\ &\vee (Q_1 \wedge Q_2 \wedge \ldots \wedge Q_{n-1} \wedge P_n)\\ &\vee (Q_1 \wedge Q_2 \wedge \ldots \wedge Q_n)\end{aligned} \tag{2}$$

Hence, if an specification presents the following schemas:

$$A == [D_A \mid P_A]$$
$$B == [A \mid P_B]$$

then Fastest first writes P_A and P_B in DNF. Then, it simply calculates the first *MDT* terms of B's DNF by applying (2), where *MDT* is a configuration variable. That is, predicates that do not reference other schemas are written in DNF by following the classic algorithm; but if a schema is unfolded inside another schema, only the first *MDT* terms of its DNF are generated. The rationale behind this heuristic lays in two observations: (a) predicates explicitly written in the predicate part of an schema tend to be small, thus keeping their DNF manageable; and (b) the explosion of terms tend to appear when included or conjugated schemas are unfolded inside other schemas. Clearly, if the DNF has more then *MDT* terms some test cases will be missed but at least some of them will be present. In the context of MBT, this is a reasonable solution compared to not calculating the DNF at all: at the end this is testing, and not formal verification.

On the other hand, if D is:

$$D == A \vee B$$

then B's DNF is calculated as above, but D will be automatically in DNF when both A and B are in that form. Obviously, if schema E is equal to the schema expression $A \Rightarrow B$, it is first rewritten as $A \vee \neg B$.

Fastest always applies DNF as the first testing tactic, regardless of the user choice. Once the predicate of an operation is in DNF, it is easy to calculate its precondition. Simply, erase those atomic predicates that have a primed or an output variable, and the resulting predicate will be the precondition of the operation. Although in Z, the precondition of a schema is another schema Fastest works at the predicate level. In Fastest, preconditions are used only to define DNF partitions and the VIS of the corresponding operations.

DOI: 10.1002/stvr

```
\begin{partition}{\cap}{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{partition}
```

Figure 4. Standard partition for $S \cap T$.

4.2. *Managing Standard Partitions*

Since Standard Partitions (SP) is a general tactic that can be applied to a number of mathematical operators, Fastest has a configuration file where users can define or modify partitions for all the operators they need. Figure 4 shows the standard partition for the $\cap$ operator delivered with the tool—which is the one originally proposed by Stocks and Carrington [3]. As the reader may note, users write standard partitions in LaTeX mark-up. Fastest includes standard partitions for $\cap$, $\cup$, $\setminus$, $\oplus$, $\in$, $\notin$, $\lhd$, $\unlhd$, $\rhd$, the cardinality operator and the arithmetic operators.

The authors of the TTF proposed a testing tactic, called Subdomain Propagation, consisting in the combination or propagation of two or more standard partitions, appearing in the definition of a complex mathematical operator, to produce a new standard partition for that complex operator [3]. The classical example is $R \oplus G$ since it is defined as $(\mathrm{dom}\, G \unlhd R) \cup G$. Then the standard partitions for $\unlhd$ and $\cup$ are combined to define the standard partition for $\oplus$. The propagation has to be done only once. Therefore, once the propagation was performed the user can add the new partition to the configuration file making it available from then on. This is one of the main reasons for which an easily extensible mechanism was designed for this key testing tactic.

4.3. *Some New Testing Tactics*

Some new testing tactics have been defined and implemented to be able to generate test specifications when some Z constructs are used or when some implementation details need to be considered. In the sections below these new testing tactic are explained and discussed.

4.3.1. Free Types. If an operation declares a variable $v : T$ where T is a free (enumerated) type, then the Free Types (FT) testing tactic generates $\#T$ test specifications whose characteristic predicates are $v = t_i$ for i in $1 \,..\, \#T$, where t_i is one of the elements of T.

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. For example, the following type lists the hardware signals that a satellite computer may receive [31]:

$$HSIGNAL ::= RP \mid AEPPS1 \mid DEPPS1 \mid AEPPS2 \mid DEPPS2 \mid EXEC \mid NONE$$

Besides, there is an operation, *ReceiveHardwareSignal*, declaring an input variable, $s?$, of that type. Therefore, if it is necessary to test for the correct reception of all the hardware signals, FT can be applied to $s?$ because in this way Fastest would generate seven test specifications, one for each signal. For instance Fastest generates the following two test specifications:

$$ReceiveHardwareSignal_1^{FT} == [srv : SRVTYPE;\ s? : HSIGNAL \mid srv = HS \land s? = RP]$$
$$ReceiveHardwareSignal_2^{FT} == [srv : SRVTYPE;\ s? : HSIGNAL \mid srv = HS \land s? = AEPPS1]$$

These test conditions cannot be generated by applying the testing tactics originally proposed for the TTF.

DOI: 10.1002/stvr

4.3.2. Dealing with set extensions. Say a company has the following departments:

$$DEPT ::= sales \mid customers \mid production \mid administration$$

And assume this company pays an annual bonus of 10% of the salary to those employees in *sales* and *production* but nothing to the others. Then, an excerpt of the formal specification for an operation calculating the bonus for an employee may be:

BonusOk

$\Xi Company$
$e? : EMPLOYEE$
$b! : \mathbb{Z};\ ret! : MSG$

$dept\ e? \in \{sales, production\}$
$b! = salary\ e? * 0.1$
$ret! = ok$

BonusE

$\Xi Company$
$e? : EMPLOYEE$
$b! : \mathbb{Z};\ ret! : MSG$

$dept\ e? \notin \{sales, production\}$
$ret! = err$

$$Bonus == BonusOk \lor BonusE$$

By applying DNF it is possible to generate the following test specifications:

$$Bonus_1^{DNF} == [Bonus_{VIS} \mid dept\ e? \in \{sales, production\}]$$
$$Bonus_2^{DNF} == [Bonus_{VIS} \mid dept\ e? \notin \{sales, production\}]$$

However, what if fetching the salary of a given employee involves different code depending on his/her department—for instance, each department might have its own database. Therefore, it would be advisable to test the bonus calculation for at least one employee in *sales*, another in *production* and yet another in any other department. More formally, the following test specifications would be a good choice:

$$Bonus_1^{ISE} == [Bonus_{VIS} \mid dept\ e? = sales]$$
$$Bonus_2^{ISE} == [Bonus_{VIS} \mid dept\ e? = production]$$
$$Bonus_3^{ISE} == [Bonus_{VIS} \mid dept\ e? \notin \{sales, production\}]$$

partitioning $Bonus_3^{ISE}$ in two is not required since the code for checking the department of a given employee should be unique. Hence, if FT is applied to *dept e?* unnecessary test cases would have been generated. Then, FT is not the right testing tactic neither.

So, for these situations Fastest defines and implements a new testing tactic called In Set Extension (ISE). It applies to operations including predicates of the form $expr \in \{expr_1, \ldots, expr_n\}$. It generates $n+1$ test specifications such that $expr = expr_i$, for i in $1\,..\,n$, and $expr_i \notin \{expr_1, \ldots, expr_n\}$ are their characteristic predicates. Note that it produces exactly the required test specifications for *Bonus*. One might be tempted to omit the last characteristic predicate because it contradicts the predicate appearing in the operation. However, observe that ISE requires the operation having such a predicate, and not necessarily the test specification being partitioned. Therefore, omitting that characteristic predicate would lead, in some cases, to non partitioning the current test specification.

Besides, Fastest defines and implements two more testing tactics to deal with other expressions involving set extensions. Proper Subset of Set Extension (PSSE) uses the same concept of ISE but applied to set inclusions like $expr \subset \{expr_1, \ldots, expr_n\}$. It generates $2^n - 1$ test specifications whose characteristic predicates are $expr = A_i$ with $i \in 1\,..\,2^n - 1$ and $A_i \in \mathbb{P}\{expr_1, \ldots, expr_n\} \setminus \{\{expr_1, \ldots, expr_n\}\}$. Subset of Set Extension (SSE) is similar to PSSE but it applies to predicates of the form $expr \subseteq \{expr_1, \ldots, expr_n\}$. Given these tactics can generate a large number of test specifications they can be weakened by applying some sort of combinatorial testing strategy like pair-wise or *n*-wise testing [32], although, so far, Fastest does not provide an automatic reduction.

DOI: 10.1002/stvr

4.3.3. Numeric Ranges. Assume that *Bonus* is implemented in the C programming language [33]. Say, *salary* has type $EMPLOYEE \pfun \mathbb{N}$. Now, think that at implementation level the salary of each employee is stored in a variable of type `short`. Since the semantics of `short` in C is quite different from the semantics of $\mathbb{N}$ in the Z notation, it would be wise to test whether the implementation properly handles salaries out of range. In effect, a `short` variable in a C program ranges over $[SHRT\_MIN, SHRT\_MAX]$, and the semantics for a possible overflow is undefined [33]. How the implementation handles salaries over $SHRT\_MAX$? To answer this question test specifications like the following are needed:

$$Bonus_1^{NR} == [Bonus_{VIS} \mid salary\ e? < SHRT\_MIN]$$
$$Bonus_2^{NR} == [Bonus_{VIS} \mid salary\ e? = SHRT\_MIN]$$
$$Bonus_3^{NR} == [Bonus_{VIS} \mid SHRT\_MIN < salary\ e? < SHRT\_MAX]$$
$$Bonus_4^{NR} == [Bonus_{VIS} \mid salary\ e? = SHRT\_MAX]$$
$$Bonus_5^{NR} == [Bonus_{VIS} \mid salary\ e? > SHRT\_MAX]$$

For these situations Fastest defines and implements a new testing tactic named Numeric Ranges (NR). With this tactic the user can bind an ordered list of numbers, $\langle n_1, \ldots n_k \rangle$, to an arithmetic expression, *expr*, in such a way that, when the tactic is applied, it generates $2 * k + 1$ test specifications characterized by the following predicates: $expr < n_1$, $expr = n_1$, $n_1 < expr < n_2$, ..., $expr = n_i$, $n_i < expr < n_{i+1}$, $expr = n_{i+1}$, ..., $expr < n_k$, $expr = n_k$ and $n_k < expr$.

This tactic is a key to bridge the gap between the specification and the implementation regarding numeric variables. No other testing tactic can generate such test specifications. Note that it provides a non-functional coverage, while all the other testing tactics and the specification itself describe functional aspects of the system. Given that at the end of the day, all the implementation limits are numbers, this tactic can be applied to test for a range of different possible overflows.

4.3.4. Mandatory Test Set. Say that it is important to be sure that the program calculates the correct bonus for these three key employees regardless of their department:

$$\mid ceo, cto, cfo : EMPLOYEE$$

Furthermore, say it is still important to test for a regular employee in each department as in Section 4.3.2. Then, test specifications such as the following are needed:

$$Bonus_1^{MTS} == [Bonus_{VIS} \mid e? = ceo]$$
$$Bonus_2^{MTS} == [Bonus_{VIS} \mid e? = cto]$$
$$Bonus_3^{MTS} == [Bonus_{VIS} \mid e? = cfo]$$
$$Bonus_4^{MTS} == [Bonus_{VIS} \mid e? \notin \{ceo, cto, cfo\}]$$

Hence, Mandatory Test Set (MTS), a tactic defined for these cases, can be applied. With MTS the user can bind a set of constants, $\{v_1, \ldots, v_n\}$ to an expression, *expr*, in such a way that, when the tactic is applied, it generates $n + 1$ test specifications characterized by the following predicates: $expr = v_i$ for all i in $1 \ldots n$, and $expr \notin \{v_1, \ldots, v_n\}$.

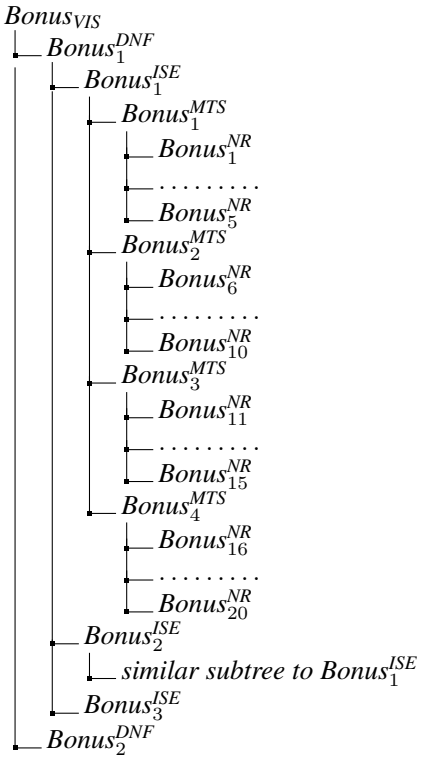
Sets are heavily used in Z specifications. Engineers might identify particular values with which to test the implementation. MTS is the indicated testing tactic in these situations. No other testing tactic can provide such a particular coverage.

4.3.5. Assembling it all together. This section is closed by showing how these testing tactics can be applied together thus building the testing tree depicted in Figure 5a. If they are combined in a different way the resulting testing tree will be different, but the leaves will be essentially the same. In Figure 5b some representative leaves are shown to make it clear how the conditions of each testing tactic are conjoined together producing test specifications that test many features at the same time.

Note that it makes no sense to populate the tree below $Bonus_3^{ISE}$ since this test specification is inconsistent because its predicate is:

$$dept\ e? \in \{sales, production\} \land dept\ e? \notin \{sales, production\}$$

DOI: 10.1002/stvr



(a) Testing tree for *Bonus*

$$Bonus_5^{NR} == [Bonus_{VIS} \mid dept\ e? \in \{sales, production\} \land dept\ e? = sales \land e? = ceo \land salary\ e? > 32767]$$

$$Bonus_9^{NR} == [Bonus_{VIS} \mid dept\ e? \in \{sales, production\} \land dept\ e? = sales \land e? = cto \land salary\ e? = 32767]$$

$$Bonus_{14}^{NR} == [Bonus_{VIS} \mid dept\ e? \in \{sales, production\} \land dept\ e? = sales \land e? = cfo \land salary\ e? = 32767]$$

$$Bonus_{18}^{NR} == [Bonus_{VIS} \mid dept\ e? \in \{sales, production\} \land dept\ e? = sales \land e? \notin \{ceo, cto, cfo\} \land 32767 < salary\ e? < 32767]$$

(b) A sample of leaves

Figure 5. A possible testing tree for *Bonus* and some of its leaves.

Other contradictions may appear in the tree depending on the department of *ceo*, *cto* and *cfo*. Section 5 describes how Fastest handles with contradictory test specifications.

4.4. Adding New Testing Tactics

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

- `List<TClass> applyTactic(TClass tClass)`. Applies this tactic to the specified test specification and returns the list with the resulting test specifications.
- `boolean parseArgs(String str)`. Parses the parameters of this tactic.

The most complex issue regarding the implementation of a new testing tactic is that programmers need to learn the annotated syntax tree built by CZT after parsing and type-checking the specification and other key data structures defined in Fastest.

5. PRUNING TESTING TREES

Due to the peculiarities of the TTF, there might be leaves from which it is impossible to get a test case because either their predicates are contradictions or they contain some undefined mathematical expression. Then, these leaves should be pruned from the testing tree prior to start the process of deriving test cases.

The problem of pruning inconsistent test specifications is as important as the problem of finding test cases from satisfiable test specifications, due to two reasons. First, the TTF tends to yield a large number of unsatisfiable test specifications. Second, since the problem of determining the set

DOI: 10.1002/stvr

of satisfiable test specifications is undecidable, any algorithm for finding test cases will yield an incomplete answer. Therefore, it is important to complement that algorithm with a method to find unsatisfiable test specifications, which, by the way is undecidable too.

After implementing the basic features of Fastest it was applied it to some of the case studies listed in Table II at page 28 [10]. Three facts were quickly discovered: (a) TTF's testing trees tend to have a large number of unsatisfiable test specifications; (b) all of them correspond to a few dozens of trivial mathematical contradictions; and (c) new projects or new testing tactics might produce new kinds of contradictions. A theorem prover was used to eliminate inconsistent test specifications but the results were not entirely satisfactory (see section 9). Then, a method specifically tailored to the TTF was designed and evaluated.

Since a complete solution is impossible, the method presented here *works in practice*, although it might not be sophisticated nor elegant. In this context, to "work in practice" means that at least 80% of the unsatisfiable test specifications appearing in *real specifications* should be pruned with minimum user intervention; it means an *engineering* or statistical solution, not necessarily a completely formal solution—after all, it is a testing method. If the method would require an action from the user to eliminate more than the 20% of the unsatisfiable test specifications, then some theorem prover, like Z/EVES [34], would yield more or less the same results making it unnecessary to develop a new method (see Section 9). Furthermore, if the method could be improved as new projects were executed, then the "work in practice" criteria might be reached as times passes and more users work with the tool.

This method was extensively described in [11]. In Section 5.1 only its most salient features are shown and in Section 5.2 it is shown how users should use it to make it more effective as more projects are run.

5.1. *Introduction to the Pruning Method Implemented in Fastest*

Fastest applies DNF by default thus making all the test specifications consist of conjunctions of atomic predicates—an atomic predicate (also known simply as an atom) is a predicate with no deeper propositional structure, that is, a predicate that contains no logical connectives or equivalently a predicate that has no strict subpredicates; atoms are thus the simplest well-formed predicates of the Z notation. The other testing tactics implemented in Fastest conjoin more atomic predicates to those already appearing in test specifications. Hence, the leaves of any testing tree are conjunctions of atomic predicates. Therefore, a leaf is inconsistent if and only if: (i) there exist atomic predicates p and $\neg\, p$ in it, (ii) there is some mathematical contradiction between two or more atomic predicates, or (iii) there is some undefined or partially-defined mathematical expression—i.e a mathematical expression whose value is not always defined [21, 34]. The first kind is easy to deal with, the second and third kinds are the core of the problem. Since our method treats contradictions and undefined terms in the same way, the exposition will be simplified by considering only contradictions.

5.1.1. Kinds of contradictions in test specifications. Before introducing the pruning method, some unsatisfiable leaves generated for a couple of operations are analysed to have an idea of what are typical contradictions. A similar analysis for over 2,000 test specifications of eleven different experiments has been performed (see section 9). Table I shows the unsatisfiable test specifications of *NewClient* eliminated in the example developed in Section 2, and some of the *Withdraw* operation, of the same example. The specification of *Withdraw* and a description of how test specifications were generated for it, are not included in this paper because they would not add any substantial information.

As it can be seen, all the contradictions are rather simple contradictory facts of set theory or integer arithmetic. When other experiments using other mathematical theories included in the Z mathematical toolkit (ZMT) [35] are considered, contradictions relative to those theories might be found [11]. Nevertheless, they are again rather simple contradictory facts of those theories. Hence, Fastest simply detects these mathematical contradictions.

DOI: 10.1002/stvr

Table I. Contradictions appearing in test specifications of *NewClient* and *Withdraw*.

Test Specification	Contradiction
$NewClient_1^{SP}$	$\{u? \mapsto name?\} = \varnothing$
$NewClient_3^{SP}$	$\{u? \mapsto name?\} = \varnothing$
$NewClient_5^{SP}$	$clients \neq \varnothing \land clients \subset \{u? \mapsto name?\}$
$NewClient_6^{SP}$	$u? \notin \mathrm{dom}\, clients \land \{u? \mapsto name?\} \subset clients$
$NewClient_7^{SP}$	$u? \notin \mathrm{dom}\, clients \land clients = \{u? \mapsto name?\}$
$NewClient_8^{SP}$	$u? \notin \mathrm{dom}\, clients \land clients \cap \{u? \mapsto name?\} \neq \varnothing$
$Withdraw_{12}^{NR}$	$m? \leq 0 \land m? = 100{,}000$
$Withdraw_{13}^{NR}$	$m? \leq 0 \land 100{,}000 < m?$
$Withdraw_{14}^{NR}$	$m? \leq 0 \land m? = 1{,}000{,}000$
$Withdraw_{15}^{NR}$	$m? \leq 0 \land 1{,}000{,}000 < m?$

```
\begin{theorem}{Reflexivity}{x, y: X}
x \neq y \\
x = y
\end{theorem}
```

ETheorem Reflexivity $[x, y : X]$

$$x \neq y$$
$$x = y$$

Figure 6. A typical elimination theorem: source code (above) and pretty-printed (below).

It is worth to mention, that some of these contradictions depend on the particular semantics of the ZMT, which is different from the semantics of the theories implemented by some theorem provers—for instance in The Coq Proof Assistant functions are not sets of ordered pairs [36]. This situation implies that it is not a trivial task to find an existing tool—either an SMT solver or a theorem prover—to solve the pruning problem for the Z notation.

5.1.2. Detecting mathematical contradictions. Fastest provides a command named `prunett` that analyses the predicate of each leaf in a testing tree to determine if the predicate is unsatisfiable or not. The most important aspect of the algorithm is a library of so called *elimination theorems* each of which represents a family of contradictions. This library can be extended by users by simply editing a text file. For example, Figure 6 depicts a typical elimination theorem included in the library.

Essentially, the method tries to match each leaf in a testing tree against each and every elimination theorem in the library. If a match is found then the leaf is pruned, otherwise nothing is done. It is important to remark that the method does not make any kind of deduction as theorem provers do. This means that if there is an elimination theorem T of which C is a matching contradiction and C' is another contradiction deducible from C, then if C' is not an instance of T, a new elimination theorem must be added in order to prune C'.

Elimination theorems are written in Standard Z LaTeX (SZL) [9]. Each elimination theorem has a set of formal parameters enclosed in square brackets. The parameters must be any legal Z declaration of variables, optionally preceded by the reserved word `\const`. If a parameter is preceded by `\const` it means that Fastest will replace it only by constants of the corresponding type. `\const` applies only to parameters of type $\mathbb{Z}$, $\mathbb{N}$ or any enumerated type (i.e. free types without induction). When an elimination theorem contains two or more constant parameters, they are replaced only by

DOI: 10.1002/stvr

different literals—this is encoded in the pruning algorithm. For instance, the library contains the following elimination theorem:

ETheorem ExcludedMiddle $[x, \text{const } y, \text{const } z : X]$

$x = y$
$x = z$

which is applied only with $y \neq z$. For example, ExcludedMiddleEq(n,1,3), but never with something like ExcludedMiddleEq(n, 3,3) nor ExcludedMiddleEq(n,*count*,1).

The predicate of an elimination theorem must be a conjunction of atomic predicates. An atomic predicate in an elimination theorem must be any legal Z atomic predicate using the standard symbols of Z, the names of the formal parameters or the reserved words `\sw`, `\anything`, `\se` and `\eval`. Currently, the method does not support user-defined symbols.

`\sw` is pretty-printed as "somewhere" (without the quotes). It takes a parameter consisting of a valid SZL token. For instance, the library contains the following elimination theorem:

ETheorem BasicUndefinition $[f : X \pfun Y;\ x : X]$

$x \notin \text{dom} f$
somewhere$(f\ x)$

somewhere(*string*) is rather similar to the regular expression $*string*$—for a full description of the semantics see [23]. When the algorithm finds such a directive it tries to match the regular expression in any of the atomic predicates of a test specification's predicate. In particular, the use of `\sw` in BasicUndefinition is useful to detect undefined expressions.

`\anything` is pretty-printed as "anything" (without the quotes). It is equivalent to the regular expression that matches any string ($*$). Two or more occurrences of this directive can match different strings. For example, the following theorem uses this directive:

ETheorem SetNotASeq $[s : \text{seq}\, X;\ n : \mathbb{N}]$

$n = 0$
$s \neq \{\}$
$\text{dom}\, s = \text{dom}\{i : 1 \mathinner{..} \text{anything} \bullet i + n - 1 \mapsto \text{anything}\}$

`\se` is pretty-printed as "se" (without the quotes). It takes a valid SZL token as a parameter. This directive is expanded to any set extension containing the real parameter for which the formal parameter was replaced.

`\eval` is pretty-printed as "eval" (without the quotes). It takes a constant Boolean expression and returns *true* or *false*. A constant Boolean expression is a Boolean expression using parameters preceded by `\const`, $\mathbb{Z}$ literals, Z operators or the literals of enumerated types. The following elimination theorem uses this directive:

ETheorem RangeNotEmpty $[n, \text{const } N, \text{const } M : \mathbb{N}]$

eval$(N \leq M)$
$n + N \mathinner{..} (n + M) = \{\}$

This sentence evaluates the Boolean expression by calling ZLive [28]. If the result is *true* and all of the other conjuncts of the elimination theorem are found in the test specification, then the test specification is pruned. If the Boolean expression evaluates to *false*, the test specification is not pruned.

Fastest applies equivalence and subtyping rules to elimination theorems whenever possible. Equivalence rules can be extended by users like elimination theorems, but Fastest always applies the rules listed in Table 7a. Equivalence rules, for instance, make it unnecessary to write the following elimination theorem:

ETheorem NotInEmptyDom_Silly $[x : X;\ R : X \rel Y]$

DOI: 10.1002/stvr

Integers	$n < m$	$m > n$
	$n > m$	$m < n$
	$n \leq m$	$m \geq n$
	$n \geq m$	$m \leq n$
Sets	$A \cap B$	$B \cap A$
	$A \cup B$	$B \cup A$
All types	$x = y$	$y = x$
	$x \neq y$	$y \neq x$

(a) Equivalence rules.

Type	Subtype
$X \leftrightarrow Y$	$X \leftrightarrow Y, X \nrightarrow Y, X \rightarrow Y, \operatorname{seq} Y$
$X \nrightarrow Y$	$X \nrightarrow Y, X \rightarrow Y, \operatorname{seq} Y$
$\mathbb{Z}$	$\mathbb{Z}, \mathbb{N}$

(b) Subtyping rules

Figure 7. Fastest applies by default these equivalence and subtyping rules.

$x \in \operatorname{dom} R$
$\operatorname{dom} R = \{\}$

because the library already contains:

ETheorem NotInEmptyDom $[x : X;\ R : X \leftrightarrow Y]$
$x \in \operatorname{dom} R$
$R = \{\}$

and the equivalence rule $R = \{\} \Leftrightarrow \operatorname{dom} R = \{\}$. Equivalence rules are lists of atomic predicates which should be equivalent to each other.

Fastest also applies some simple subtyping rules when it substitutes the formal parameters of an elimination theorem or equivalence rule by actual parameters. A subtyping rule determines whether a type or set is a subtype of another type or set[†]. For instance $X \rightarrow Y$ is a subtype of $X \nrightarrow Y$ which in turn is a subtype of $X \leftrightarrow Y$. The subtyping rules applied by Fastest are listed in Table 7b. Therefore, NotInEmptyDom above applies not only to binary relations but to functions and partial functions as well.

As it can be seen, the method proposed here for pruning inconsistent test specifications features some techniques that make it possible to generalize each elimination theorem. In this way, a single elimination theorem is useful to prune many inconsistent test specifications.

5.2. *Maintaining the Elimination Theorem Library*

The philosophy behind the pruning method implemented by `prunett` is that users should add an elimination theorem every time they find an inconsistent leaf in a testing tree—because this will help them in future projects since similar inconsistent leaves will be pruned automatically. This section shows how to do that considering the unsatisfiable test specifications of Table I. It is assumed that the library does not contain elimination theorems specifically added to prune those test specifications.

Consider $NewClient_1^{SP}$ and $NewClient_3^{SP}$. The first choice for an elimination theorem might be:

ETheorem NoCorrect $[u? : UID;\ name? : NAME]$
$\{u? \mapsto name?\} = \varnothing$

but this elimination theorem is unnecessarily restrictive since $\{u? \mapsto name?\}$ is not empty not because it contains an ordered pair of that particular type but because it contains an element. Furthermore, $\{u? \mapsto name?\}$ is not only a set but, more specifically, a set extension. Since any set extension, of any type, is non-empty, then the right choice is:

ETheorem SetExtensionIsNotEmpty $[x : X]$
$\operatorname{setExtension}(x) = \varnothing$

[†]Type and sets are often used interchangeably in Z

DOI: 10.1002/stvr

In other words, this elimination theorem will prune test specifications containing an atomic predicate stating that a set extension is equal to the empty set regardless of its type and the number of elements belonging to it.

$NewClient_5^{SP}$ is unsatisfiable because only an empty set can be strictly contained in a singleton. Therefore, it is convenient to generalize the corresponding elimination theorem to any non-empty set strictly contained in a singleton. Hence, the elimination theorem is as follows:

ETheorem NotSubsetOfSingleton $[A : \mathbb{P}\, X;\ x : X]$

$A \neq \{\}$
$A \subset \{x\}$

Subtyping rules are good to state the elimination theorems for $NewClient_6^{SP}$ and $NewClient_7^{SP}$ since their contradictions are true not only for partial functions (*clients*) but for binary relations in general. This means that the type of one of its formal parameters should be $X \leftrightarrow Y$ instead of $X \nrightarrow Y$, because if the former is used then Fastest, by means of its subtyping rules, will apply it to elements of the latter type, too. Therefore, the elimination theorems are, respectively:

ETheorem SingletonMappletNotSubsetRel $[R : X \leftrightarrow Y;\ x : X;\ y : Y]$

$x \notin \mathrm{dom}\, R$
$\mathrm{setExtension}(x \mapsto y) \subset R$

ETheorem SingletonMappletNotEqualRel $[R : X \leftrightarrow Y;\ x : X;\ y : Y]$

$x \notin \mathrm{dom}\, R$
$\mathrm{setExtension}(x \mapsto y) = R$

$NewClient_8^{SP}$ is interesting because the following equivalence rules:

EquivRule MembershipEquivalences $[A : \mathbb{P}\, X;\ x : X]$

$x \in A$
$\{x\} \cap A \neq \{\}$

EquivRule NotMembershipEquivalences $[A : \mathbb{P}\, X;\ x : X]$

$x \notin A$
$\neg\ \{x\} \subseteq A$

make it unnecessary to define a new elimination theorem since the test specification will be pruned by an elimination theorem that is already present in the library:

ETheorem BasicMembershipContradiction $[A : \mathbb{P}\, X;\ x : X]$

$x \in A$
$x \notin A$

In this case `prunett` applies: (i) MembershipEquivalences to $clients \cap \{u? \mapsto name?\} \neq \varnothing$ making it equivalent to $\{u? \mapsto name?\} \in clients$; (ii) NotMembershipEquivalences to $\neg\ \{u? \mapsto name?\} \subseteq clients$ (in $NewClient_8^{SP}$ at page 7) making it equivalent to $\{u? \mapsto name?\} \notin clients$; and (iii) BasicMembershipContradiction to the two preceding predicates.

For the remaining four test specifications of Table I only two elimination theorems are needed. The first one prunes $Withdraw_{12}^{NR}$ and $Withdraw_{14}^{NR}$:

ETheorem ArithmIneq1 $[\mathrm{const}\, N, \mathrm{const}\, M : \mathbb{Z};\ n : \mathbb{Z}]$

$\mathrm{eval}(N < M)$
$n \leq N$
$M = n$

DOI: 10.1002/stvr

Note that it has been generalized by replacing 0, 100000 and 1000000 by constant formal parameters that must satisfy $N < M$. The second elimination theorem is very similar:

ETheorem ArithmIneq2 [const N, const $M : \mathbb{Z}$; $n : \mathbb{Z}$]

$\text{eval}(N \leq M)$
$n \leq N$
$M < n$

Both elimination theorems are necessary because the pruning method does not include any kind of arithmetic satisfiability nor it performs term rewriting. As it has been said above, the method works solely by parametrized pattern-matching. Therefore, it is not necessary to implement any complex SMT functionality covering all the Z theories because they can be encoded in the library. Clearly, the pruning method takes advantage of the peculiarities of the TTF. At the same time, it proved to be effective and efficient in practice as shown in Section 9.

5.3. Soundness

The pruning method is sound if the following two conditions hold:

1. The pruning algorithm is correctly implemented.
2. The library contains only contradictions.

In general, these two points aim at avoiding the elimination of satisfiable test specifications, which would imply not testing the SUT as expected. Fastest has been tested with more than 2,000 test specifications of more than ten different Z specifications besides a number of specific test cases. Of course this is not a proof of correctness but gives a reasonable level of assurance.

On the other hand, as Cristiá et al. have described [11], a theorem prover can be used to guarantee the second point. In effect, they have certified with Z/EVES 50 of the 52 elimination theorems contained in the library at the moment the experiments shown in Section 9 were performed. The remaining two were not certified because they use the reserved work `\sw` which is not easy to represent in Z—this issue is currently being investigated. In future releases, Fastest could be extended to authorize the addition of a new elimination theorem only if a certificate is attached to it.

6. FINDING ABSTRACT TEST CASES WITH FINITE MODELS

Once test specifications have been generated it is necessary to derive at least one abstract test case for each of them. Within the TTF finding an abstract test case for a given test specification means to find a witness satisfying its predicate. In turn, this means to find a constant value for each free variable in the predicate such that it contributes to satisfy it. Usually free variables range over infinite types, making the problem undecidable. As with pruning, the rich mathematical theories defined in the ZMT and their specific semantics, makes it particular difficult to use existing tools to solve the problem. For instance, the most obvious solution would have been to rely on a SMT solver [37, 38] but: (i) it was impossible to find an existing SMT solver covering all or a significant portion of the ZMT; and (ii) either implementing a SMT solver from scratch or trying to codify the ZMT in an existing SMT are research projects in their own not directly related with our interests on MBT.

Then, it has been decided to devise, once again, a simple method meeting a "work in practice" criteria. In other words, a method which should automatically find an abstract test case for at least 80% of the satisfiable test specifications appearing in *real specifications*, in a reasonable time, was envisioned. A less ambitious limit would put the whole process based on the TTF at risk because manual testing, or other testing techniques as well, would become more appealing since one of the greatest advantages of MBT is its degree of automation—the simulation made by Utting and Legeard in [1, pages 35–40] was the main guideline in this respect.

The method to find abstract test cases from test specifications implemented in Fastest consists of building a finite model for each test specification by calculating the Cartesian product between a very

DOI: 10.1002/stvr

small finite set of values for each variable in the VIS. Later, Fastest evaluates the test specification for some elements in the finite model. Although this algorithm might appear inefficient and is certainly inelegant, it has proved to meet the "work in practice" criteria, as it is shown in Section 9. Due to the undecidable nature of the problem, every time Fastest tries to find an abstract test case from a given test specification, only one of the following can happen:

- Some element in the finite model satisfies the test specification, then the desired abstract test case has been found.
- There is no element in the finite model satisfying the test specification because it is a contradiction—and was not pruned in the previous step.
- There is no element in the finite model satisfying the test specification, but it is not a contradiction.

Users have to assist the tool in the last two cases. Either they ought to add an elimination theorem (Section 5), or they have to help Fastest in finding the abstract test case, as will be shown shortly.

6.1. Default Finite Sets

One of the key decisions that makes the method work in practice is what finite sets Fastest should consider for each variable in a test specification. After applying the first versions of the tool to some real case studies, a set of heuristics that make it possible to find a very high number of test cases was defined. These heuristics are the following:

- There is a configuration variable, *FSS*, whose value sets the size of the finite sets for basic types, $\mathbb{Z}$ and $\mathbb{N}$. *FSS* must be strictly positive.
- The finite sets for types $\mathbb{N}$ or $\mathbb{Z}$ are built from the first *FSS* numerical constants appearing in the test specification. If there are no such constants then $0 \mathinner{..} FSS - 1$ is chosen for $\mathbb{N}$ and $-(FSS \operatorname{div} 2 + (FSS \bmod 2 - 1)) \mathinner{..} (FSS \operatorname{div} 2)$ for $\mathbb{Z}$.
- The finite sets for enumerated types are their elements. For example, if a test specification declares a variable of type *HSIGNAL* (page 12), then its finite set will be all the elements of *HSIGNAL*.
- The finite sets for basic types are built by generating *FSS* constant names of each type. For example, if *CHAR* is a basic type, Fastest assumes the existence of $char_1, \ldots, char_{FSS} : CHAR$, and then $\{char_1, \ldots, char_{FSS}\}$ becomes the finite set for any variable of type *CHAR*.
- If a variable declared in the *VIS* does not appear in the predicate of a test specification, 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 specification 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 the types that result from applying a type constructor—i.e. $\times$, $\mathbb{P}$, $\pfun$, etc.—to other types, are built recursively from the finite sets considered for its arguments.

6.2. Evaluating Test Specifications on Finite Models

Once the finite model for a given test specification has been built, the algorithm searching for abstract test cases takes each element of this model and evaluates the test specification on it—the details of this iteration are more complex but they are outside the scope of this paper. If the *IS* of a test specification is $[x_1 : T_1, \ldots, x_n : T_n]$ then its finite model, *FM*, has the same type. Let *P* be the predicate of a test specification, then it depends on $x_1, \ldots, x_n$. Let *c* be an element of *FM*. The evaluation of *P* on *c* comprises two steps:

1. Substitute each free variable in *P* by the corresponding constant of *c*, i.e. calculate $P(x_1/c_1, \ldots, x_n/c_n)$. This yields a predicate with no free variables.
2. Call the method `evalPred` passing it as a parameter the result of the previous step. `evalPred` is a method of the class

DOI: 10.1002/stvr

`net.sourceforge.czt.animation.eval.ZLive` of the ZLive component of the CZT project [28]. This method evaluates a constant Z predicate returning *true* or *false*.

6.3. Taking Advantage of the Testing Tree Structure

As it has been shown in section 2, leaves in a testing tree share with their parent node some predicates—for an instance, compare the predicates of $NewClient_1^{DNF}$ with those of its children at page 7. Taking advantage of this commonality improves the algorithm since a smaller finite model can be used as the initial one. Let t be a leaf in a testing tree and t' its parent node. Say $FM(t')$ is the finite model generated by Fastest for t'. Assume $A(t')$ is the subset of $FM(t')$ that satisfy t', as returned by the algorithm. In the worse case $A(t') = FM(t')$ but in practice $A(t')$ tends to be much smaller than $FM(t')$. Therefore, Fastest runs the algorithm in two stages:

1. The algorithm is run on t' taking $FM(t')$ as the initial finite model. If $A(t') = \varnothing$, then either t is unsatisfiable or the algorithm failed to find an abstract test case for it—because t just conjoins some new atomic predicates to t'. In either case the algorithm terminates.
2. If $A(t') \neq \varnothing$ then the algorithm is run on every child, t, of t' taking $A(t')$ as the initial finite model—instead of $FM(t)$. In this stage the algorithm is called to return the first element satisfying the test specification—and not the full set of values.
   If, as usual, $A(t')$ is much smaller than $FM(t')$, then the algorithm is more efficient and as effective as taking $FM(t)$ as the initial finite model for t. On the other hand, if $A(t') = FM(t')$, then the efficiency is the same because t would have been evaluated over $FM(t')$ anyway—because t just conjoins some new atomic predicates to t'.

In this way, a satisfiability set is calculated only once for the common sub-predicate included in each leaf. If testing trees are deep and many leaves hang from the same parent node, this considerably improves the efficiency of the algorithm. This improvement takes advantage of the peculiarities of the TTF.

6.4. FSS and MAX, Two Key Configuration Variables

In section 6.1 *FSS* was introduced. The value of this variable has a great impact on the efficiency and effectiveness of the algorithm since it influences the size of the default finite sets bound to variables with complex types. For instance, the finite set bound to $f : \mathbb{Z} \pfun \mathbb{Z}$ has 40 elements if a 3 elements set is bound to $\mathbb{Z}$, it has 7 if $\mathbb{Z}$ is bound to a 2 elements set, and just 2 if only one integer number is considered. The influence is more evident if $g : \mathbb{Z} \pfun \mathbb{Z} \pfun \mathbb{Z}$ is considered because in this case sizes are: 65,641 for 3, 57 for 2 and 5 for 1.

The algorithm also relies on the value of another configuration parameter, namely *MAX*. *MAX* is the maximum number of evaluations that will be performed to search for an abstract test case of a given test specification. This parameter was introduced because test specifications declaring many variables or variables of complex types—higher order functions, for instance—may give rise to huge finite models that would take a very long time to be explored completely.

Obviously, the greater the values of *FSS* and *MAX*, the greater the chances to find an abstract test case in any test specification but the slower the algorithm. Section 9 shows the results of some experiments for different values of these parameters.

6.5. The `setfinitemodel` Command

When the algorithm fails to find an abstract test case for a satisfiable test specification, users need to help the tool to find one. Fastest allows users to define a finite model or part of it for a given test specification by issuing a `setfinitemodel` command. When `genalltca` is run again, these models are used instead of the default ones. If users define the right finite models, then all the test cases will be derived. The command receives the name of a leaf in a testing tree and a list of parameters that bind a finite set to all or some of the VIS variables or the types used in it. If some variable is not bound to a finite set with this command, then Fastest uses the default one. For

DOI: 10.1002/stvr

example, if a test specification, T, declares $x, y : \mathbb{Z}$, and its predicate is $x > 0 \land y > x$, then Fastest will not be able to find a test case for it. Hence, the following command solves the problem:

```
setfinitemodel T -fm "\num == 1 \upto 2"
```

6.6. Soundness

The soundness of this algorithm depends on the correct implementation of the `evalPred` method because if it returns *true*, when it actually should have returned *false*, then an incorrect abstract test case will be bound to a test specification; and this in turn implies not testing the SUT as expected. As it has been said, `evalPred` is part of the CZT project which has a larger user community than Fastest and many tools have been build on top of it.

On the other hand, the following two situations do not represent soundness problems: (a) the algorithm returns *false*, when it should have returned *true*; and (b) Fastest builds a wrong finite model for a test specification making it impossible to find an abstract test case. In either situation, no abstract test case will be bound to the test specification. However, Fastest provides the `certified` command which returns *true* when there is an abstract test case bound to all the leaves of all the testing trees, and *false* otherwise. Therefore, if the testing trees are not certified, users have the chance of noticing it and take an appropriate action.

7. AUTOMATIC TRANSLATION OF TEST CASES INTO NATURAL LANGUAGE

The problem of generating natural language (NL) descriptions of abstract test cases written in Z arises in the following scenario: a company developing the software for a satellite needs to verify that the implementation conforms to a certain aerospace standard describing the basic functionality of any satellite software [39]. Therefore the project started by modelling in Z the services described by the standard and then followed by using the Fastest tool to generate almost 200 abstract test cases. The customer requested to deliver a natural language description of each one of them. Clearly, trying to make the translation by hand would have been not only a source of errors, but also a technical retreat. Given the formal, structured nature of the source text, natural language generation (NLG) techniques [13] seem to be an appropriate approach to solving this problem. This work has bee described in a previous paper [12], so here only the main aspects are included.

7.1. A Template-Based NLG Solution

As a first approach, a NLG template-based method has been used [40]. First, a grammar was defined to express what is called *NL test case templates* (NLTCT). Each NLTCT specifies how a NL description is generated for the test cases of a given operation. It starts with the name of the operation. Next follows a text section, intended as a parametrized NL description of the test case, where calls to *translation rules* can be inserted as needed. Finally, all necessary translation rules are defined, by indicating what is written in replacement for a call when a certain variable in the formal description of a test case appears bound to a specific value. In this way, a different text is generated for each test case, according to the binding between values and variables that defines the test.

A parser in `awk` that takes an NLTCT and a list of Z test cases, and generates the NL description of each test case in the input, has been implemented. Figure 8 is the NLTCT for one of the Z operations of the formal specification of the satellite software. In turn, Figure 9 shows the result of applying the `awk` script to a test case with the NLTCT of Figure 8.

This first prototype showed that NLTCTs tend to be relatively small and simple, in spite of the large number of test cases. This is due to test cases combining a small set of values in many different ways. However, NLTCTs for large operations tend to become increasingly more complex, for the number of combinations grows exponentially. As a consequence, these operations require a large number of cases within translation rules and sometimes even more translation rules. This is because templates are written in terms of the values bound to variables, and not in terms of the predicates satisfied by those values, which are nonetheless available as part of the MBT approach.

DOI: 10.1002/stvr

```
operation = DumpMemoryAbsAdd

\begin{tcase}
\centerline{{\bf Test case: \ltcaseid}}

The service (6,5) will be tested in the
situation that verifies that:
\begin{itemize}
  \item the state is such that:
  \begin{itemize}
    \item the on-board system is &trule PTCr&.
    \item the service type of the telecommand
          is &trule SRVr&.
    [...]
    \item the set of starting addresses of the
          chunks of memory that have been
          requested by the ground is &trule
          ADSr&.
  \end{itemize}
  [...]
\end{itemize}
\end{tcase}

\begin{trule}{PTCr}
case processingTC
$yes :currently processing a telecommand and
      has not answered it yet
$no :not currently processing a telecommand
endcase
\end{trule}

\begin{trule}{SRVr}
case srv
$* :*
endcase
\end{trule}

\begin{trule}{ADSr}
case adds
$\emptyset :empty
$\langle 0 \rangle :the unitary sequence
                    composed of 0
endcase
\end{trule}

[...]
```

Figure 8. NLTCT for a Z operation of the aerospace standard formalization.

Test case: DumpMemoryAbsAdd_SP_7_TC

Service (6,5) will be tested in a situation that verifies that:

- the state is such that:
  - the on-board system is currently processing a telecommand and has not answered it yet.
  - the service type of the telecommand is DMAA.
  - the set of sequences of available memory cells contains only one sequence, associated to a memory ID, which has four different bytes.
  - the set of starting addresses of the chunks of memory that have been requested by the ground is empty.
- the input memory ID to be dumped is the available memory ID, the input set of start addresses of the memory regions to be dumped is the unitary sequence composed of 1, the set of numbers of memory cells to be dumped is the unitary sequence composed of 2.

Figure 9. NL description of a typical abstract test case.

8. DEGREE OF AUTOMATION AND LIMITATIONS

This section discusses the degree of automation and the limitations of Fastest in each step of the MBT process based on the TTF. This process comprises the steps listed in Section 2 and the translation of the abstract test cases into natural language (Section 7). Some of the conclusions drawn here are backed-up by the empirical assessment presented in Section 9.

1. Consider the valid input space (VIS) of each Z operation. Command `selop`.
   The user must select those schemas that correspond to the operations to be tested. Once this is done, Fastest automatically builds the VIS of each operation. The selection of schemas has been left to the user because many schemas are operations but it would be wrong to derive abstract test cases for all of them. For example, *ClientAlreadyExists* in Section 2, formally, is an operation—since it declares before and after-state variables—but users may prefer to generate test cases directly for *NewClient*, which references *ClientAlreadyExists* as well, and in doing so they will get test cases for it too.
2. Apply one or more testing tactics. Command `addtactic`.
   Fastest automatically applies DNF when the first testing tree is built. Users must give a command for each testing tactic they want to apply to each selected operation. These commands receive the name of a node in a testing tree, the name of a testing tactic and a list of parameters that depend on the testing tactic. Users can add testing tactics and then

DOI: 10.1002/stvr

build the tree or they can interleave both commands. Fastest automatically generates all the test specifications and testing trees as indicated by the testing tactics.
Since deciding which testing tactics to apply to each operation is essentially an analytical task, it has been left to be performed by an engineer. The selection of the right testing tactics are the key to a good coverage. However, as a future work, one might consider implementing heuristics that automatically applies some testing tactics depending on an static analysis of the operation. For example, if an operation declares a variable of an enumerated type then FT is applied, if it also uses, say, $\cup$ then SP is also applied, and so on.

3. Build a tree of test specifications. Command `genalltt`.
   As shown in the previous step, Fastest automatically builds testing trees. This implies that, basically, Fastest fully automates the generation of test specifications, leaving just the analytical work to the user.
4. Prune inconsistent test specifications. Command `prunett`.
   Given its nature, this step is necessarily semi-automatic. It is difficult to assess the degree of automation of this step since it depends on how many projects were previously run, because the cornerstone of the method is a user-maintained library of contradictions. Therefore, if users properly maintain this library, the method will be more automatic as more projects are run. According to the experiments shown in Section 9, users need to add elimination theorems to the library when a specification uses mathematical theories that were not used in the previous projects, or when they combine testing tactics in a new way. For example, if an operation uses $\cup$ and $\subseteq$ and the user combines SP for these operators for the very first time, then some contradictions involving these operators might appear. In this case some test specifications will not be pruned and the user will ought to add some new elimination theorems. If these elimination theorems are properly generalized, Fastest will tend to prune all the unsatisfiable test specifications of a new project where the same (syntactically) operators are used.
   Since the sets of mathematical operators and testing tactics are finite and given the generalization techniques provided by the pruning method, it can be argued that the number of elimination theorems will increase rapidly until a few dozen of projects are run, and then it will remain more or less stable. It is not hard to think of a library of elimination theorems accessible to all users world-wide, thus increasing the degree of automation of this step with the contributions of all of them.
5. Find abstract test cases. Commands `genalltca` and `setfinitemodel`.
   As with the previous step, this one is necessarily semi-automatic. The experiments described in Section 9 show that, in average, Fastest finds an abstract test case for the 80% up to the 89% of the satisfiable test specifications, in a reasonable time—depending on the value of *FSS* and *MAX* (Section 6.4).
   Once a testing tree has been built, the most promising course of action is to run `prunett` immediately followed by `genalltca`. This will yield some test specifications with their test cases and some not. Therefore, users must analyse those test specifications without test cases to see whether they are satisfiable or not. If they are unsatisfiable, then one or more elimination theorems must be added; `prunett` can be run again. If they are satisfiable, users need to issue a `setfinitemodel` command for each test specification.
   After using Fastest in several experiments and projects, it has been noted that the algorithm tends to fail in finding abstract test cases for groups of leaves. Usually, it fails due to the same reason for all of them. Therefore, possibly, `setfinitemodel` will be generalized to set a finite model for a sub-tree and not only for a single leaf. This would severely reduce the time spent in this step.
6. Translate each abstract test case into natural language.
   As with the previous two steps, this one is again necessarily semi-automatic. Users need to define a NLTCT for each operation but then all the NL descriptions are automatically generated. As shown in Section 7, writing a NLTCT involves writing the general description for a typical test case and bind a NL description for each possible value of each VIS variable.

DOI: 10.1002/stvr

9. EMPIRICAL ASSESSMENT

The main purpose of this section is to empirically assess the pruning method and the algorithm to search for test cases. This assessment has been made on the basis of applying Fastest to eleven case studies. All these experiments were conducted on the following platform: an Intel Centrino Duo of 1.66 GHz with 1 Gb of main memory, running Linux Ubuntu 10.04 LTS with kernel 2.6.32-25-generic and Java SE Runtime Environment 1.6.0_18. Fastest was run as a stand-alone application, i.e. not in client-server mode, with the following command:

```
java -Xss8M -Xms512m -Xmx512m -jar fastest.jar
```

9.1. The Case Studies

In this section a brief informal description of each case study is given. They belong to several different application domains.

SWPDC. Part of the communication protocol between two computers of a Brazilian satellite. This model has operations to load a program in the memory of one of the computers, to transmit data between the computers, to interact with some hardware devices, to dump the memory, and so on.

Aerospace standard Five of the services described in [39]. It was used as a case study by a satellite development company from Argentina to evaluate this methodology.

Plavis. A proprietary protocol for the communication between a so called experiment computer and the On-Board Data Handling (OBDH) computer, of a Brazilian scientific satellite.

Scheduler. This model was borrowed from [1]. Basically, the B model described in that book was translated into Z.

Savings accounts (three functions). Part of this model is shown in Section 2. 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. Accounts can be owned by many customers and some of their personal data are also registered. The state schema of this model is composed by three state variables of functional types.

Launcher vehicle. This is part of the on-board flight control software of a Brazilian satellite launcher vehicle. It includes processing several events that are sent by sensors during the flight of the launcher.

Security class. A security class is a computer security concept belonging to mandatory access control [41]. 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 any type.

Savings accounts (one function). This is a simplified version of a similar problem described above. The state schema of this model is composed by only one state variable of functional type. It makes sense to include both versions because it is important to assess how Fastest performs with both bigger and smaller models.

Lift. This model was borrowed from [42].

Symbol table. This model was borrowed from [3].

Pool of sensors. This is the specification of a subroutine which shall keep the highest readings of an array of sensors.

DOI: 10.1002/stvr

Table II. Complexity and size of the case studies.

CS	Case study	Real/Toy	LOZC	State	Operations	Leaves	Satisfiable
1	SWPDC	Real	1,238	18	17	201	144
2	Aerospace standard	Real	774	14	5	1226	196
3	Plavis	Real	608	13	13	229	181
4	Scheduler	Toy	240	3	10	199	34
5	Savings accounts (3)	Toy	165	3	6	105	25
6	Launcher vehicle	Real	139	4	1	180	38
7	Savings accounts (1)	Toy	171	1	5	97	25
8	Security class	Toy	172	4	7	36	20
9	Lift	Toy	152	6	3	17	16
10	Symbol table	Toy	78	1	3	26	10
11	Pool of sensors	Toy	46	1	1	15	8

Table II gives an idea of the complexity and size of each case study. **CS** is the number of case study which links this table with the following ones. **LOZC** stands for lines of Z coded in LaTeX mark-up. Columns **State** and **Operations** represent the number of state variables and operations, respectively, defined in each specification. **Leaves** and **Satisfiable** are the number of initial leaves (before pruning) of each testing tree and the number of them that are satisfiable—this was determined by manual inspection.

9.2. *Effectiveness and Efficiency of the Pruning Method*

The results presented in this section were published in a previous paper [11]; they are include hero so readers can find an integral presentation of the experiments carried on with Fastest. The philosophy behind the pruning method is that users should maintain the library of elimination theorems so the method becomes more automatic as more projects are executed. Although this feature is obvious from the description of the method, is it easy to maintain the library or is so difficult that other approaches are better options? In other words, how many elimination theorems should users add for each new project? In order to answer these questions empirically the following simulation was carried on: (i) a user that starts with a library containing only three basic elimination theorems; and (ii) then he/she runs one case study after the other, adding elimination theorems as unsatisfiable test specifications are found. In this experiment, the case studies of Table II were used starting from the simplest Toy example to the most complex of them; in a second stage, the Real ones were randomly run. The aim of the second stage was to recreate a more realistic situation.

Table IIIa summarizes these results. **ET** is the *cumulative* number of elimination theorems necessary to prune all the unsatisfiable test specifications—i.e. the difference between successive rows is the number of elimination theorems that had to be added to prune all the unsatisfiable test specifications. **Mathematical Theories** are the new mathematical theories used by each case study with respect to the previous ones. As it can be seen, the user intervention decreases as new projects are executed. Every time a model not adding new mathematical theories is loaded into Fastest, almost no new elimination theorems are needed—see the rows where column **Mathematical Theories** is empty and compare the values of column **ET** of their predecessors.

Table IV shows the results of comparing our pruning method with the Z/EVES proof assistant [34]. The intention of these experiments is to confirm that by exploiting the peculiarities of the TTF it is possible to achieve better results than with an out-of-the-shelf theorem prover, even one especially tailored to the Z notation. It must be noted that the mathematical toolkit of Z/EVES contains 565 theorems but Fastest needed just 55 elimination theorems to prune all the inconsistent test specifications of all case studies. As the reader can see, Fastest outperforms Z/EVES in all the key issues proposed for the comparison: it prunes more test specifications and takes invariably less time than Z/EVES, with notably less theorems.

In [43] the authors apply the Isabelle theorem prover to, among other things, eliminate unsatisfiable test specifications. They use an encoding of Z in Isabelle, called HOL-Z [44], to generate test specifications for the *STEAM_BOILER_WAITING* operation of the steam boiler control

DOI: 10.1002/stvr

Table III. Results of the empirical assessment of the pruning method.

(a) Ease-of-use of the pruning method.

CS	ET	Mathematical Theories
11	7	Integer inequalities
10	16	Basic set theory
9	17	
8	17	
7	20	Relational domain
4	33	Cardinality, singletons, relational range
3	38	Sequences
1	52	Integer ranges
2	52	
6	54	
5	55	

(b) Comparison of pruning between Z/EVES and Fastest.

CS	Z/EVES		Fastest	
	Pruned	Time	Pruned	Time
1	21	16m31s	56	31s
2	728	1h20m51s	856	2m18s
3	19	6m50s	50	16s
4	123	26s	160	7s
5	39	56s	48	2.6s
6	152	31s	172	8.7s
7	45	15s	75	3s
8	14	4s	16	0.8s
9	1	11s	1	0.5s
10	11	2s	16	0.7s
11	3	1s	8	0.5s

software specified in Z [45]. They recognize the need to eliminate test specifications after applying DNF and show their results in terms of the number of simplified test specifications and the computing time needed to do that for four experiments. The same experiments were performed with Fastest obtaining much better computing times—although part of this improvement surely comes from the differences on the hardware platforms—and the same number of pruned test specifications [11]. Therefore, considering just these experiments, it can be concluded that Fastest is at least as effective than the HOL-Z environment in simplifying testing trees.

The better performance of this pruning method, possibly, stems from the fact that it was born as a specialized solution for a particular problem of the TTF, while Z/EVES and HOL-Z are are aimed at much general and harder problems. Perhaps it could have been possible to encode this method in the tactic language of some conventional theorem prover—after encoding the Z notation—, but it would have required from Fastest's users to learn theorem proving techniques. This would have been another impediment for the adoption of Fastest by the industry.

9.3. Effectiveness and Efficiency of the Test Cases Generation Algorithm

This section shows an empirical assessment of the abstract test case generation algorithm introduced in Section 6. The effectiveness and efficiency of the algorithm was assessed against two magnitudes: (a) the total computing time needed to try to find a test case for each test specification, for each case study; and (b) the number of test cases that the algorithm can discover with respect to the number of satisfiable test specifications. At the same time, optimal values for *FSS* and *MAX* (Section 6.4) were sought. To do so, based on previous experience, only experiments with $FSS \in \{2, 3\}$ and $MAX \in \{100, 500, 1000\}$ were conducted. Due to space restrictions Table IV only shows the results of $FFS = 2$ and $MAX \in \{100, 500\}$. As it can be seen, the average percentage of test cases found from satisfiable test specifications meet the "work in practice" criteria. Note that the very nature of the problem posses a limit on the number of test specifications that can be satisfied by any algorithm. Likely, more sophisticated techniques might increase this percentage, but none will be able to solve always all the cases.

Finding the optimal values for *FSS* and *MAX* means to reduce the computing time to calculate abstract test cases while augmenting the percentage of solved test specifications with respect to the number of satisfiable test specifications. Therefore, the quotient between the percentage of solved test specifications and the computing time needed to run the algorithm for all the leaves—both satisfiable and unsatisfiable—was calculated. It is clear that the higher the quotient the better the combination. It turned out that 2 was the best value for *FSS* and the best values for *MAX* were 100 and 500. The combination $FSS = 2$ and $MAX = 100$ can be regarded as the best, since it is better in more case studies than the rest.

DOI: 10.1002/stvr

Table IV. The two best combinations between *FSS* and *MAX*.

	$MAX = 100$			$MAX = 500$		
CS	**Time**	**%**	**Quotient**	**Time**	**%**	**Quotient**
1	08m08s	76	13456	18m00s	92	7360
2	12m18s	56	6556	39m15s	57	2091
3	17m46s	77	6241	31m06s	84	3889
4	01m44s	94	78092	01m45s	94	77349
5	02m31s	83	47491	04m07s	83	29033
6	02m54s	71	35255	04m06s	79	27746
7	01m22s	96	101151	01m31s	100	94945
8	00m17s	80	406588	00m17s	80	406588
9	00m41s	100	210732	00m40s	100	216000
10	00m25s	100	345600	00m25s	100	345600
11	00m19s	50	227368	00m40s	75	162000
Average		80%		**Average**	85%	

10. RELATED WORK

There are a number of different MBT approaches proposed in the literature [2] and some tools advertised by both industry and the academia [46]. In this section the approach presented in this paper is compared only with the most similar of them all. First, it must be noted that Fastest is the first implementation of the TTF providing such a degree of automation. TTF's original authors implemented TinMan [47, 48] but 'it was aimed primarily at providing organization support with little support for manipulating predicates' [5].

There are other MBT methods besides the TTF available for the Z notation. Ammann and Offutt apply category–partition testing to Z models [49]; Hall generates tests by analysing the test domains of Z operations [50]; Hierons, Sadeghipour and Singh use the Z information provided in a μSZ specification to provide sequences of transitions that covers a EFSM derived from the specification [51]; Hierons also partitions a Z operation and then he derives a FSA to control how testing is performed [52]; Horcher and Peleska apply DNF to a Z operation and describe a MBT process similar to the one in Figure 1 [42]; Burton formally specifies partitioning tactics which are used to generate test specifications by applying theorem provers to Z specifications [53].

Legeard, Peureux and Utting compare the TTF and BTT—a MBT method for the B notation [54]—by applying them to the same case study [5]. The authors conclude that the generation of test cases within the TTF is a manual process. This paper presents convincing arguments that most of TTF's activities can be automated to a reasonable extent.

The satisfiability algorithm presented in Section 6 is similar in conception to approaches like the Alloy Analyzer which also defines a finite model for a given predicate and tries to see whether is it possible to satisfy it within this model or not [55]. Although Alloy was in part inspired by Z, its language and type system are different and the analyzer uses more advanced SAT techniques. As part of the future work with Fastest and the TTF it is possible to investigate the chances of integrating them with SAT or SMT solvers [37, 38], with the intention to increase the 80% limit in finding abstract test cases and to lower its computing time. SMT solvers supporting uninterpreted functions or some mathematical theory over which a shallow embedding of the ZMT can be performed, is the most promising approach to improve the search for test cases within the TTF given the number and complexity of the mathematical theories present in the ZMT. Such SMT solvers include, for instance, Z3 [56], Yices [57] and CVC3 [58]. However, it is a non trivial project to encode Z into the language of a SMT. Other approaches solve similar problems but in different contexts making them less useful for the TTF. For example: Korat is a tool that generates complex inputs that satisfies a constraint but these inputs are for Java programs [59]; and Hèam and Nicaud can automatically generate random tests for a variety of grammars [60], but being random they do not easily fit the needs regarding the satisfiability of Z predicates.

Section 7 shows how abstract test cases generated by Fastest can be translated into natural language. There have been efforts for producing natural language versions of formal specifications

DOI: 10.1002/stvr

in the past [61, 62, 63, 64, 65]. However, these solutions are highly dependant on particular aspects of the source language and do not apply directly to specifications written in Z. As far as it was investigated, no work has been done towards producing NL descriptions of Z specifications. The same holds for test cases generated using the MBT approach.

An MBT tool that is similar to Fastest is LTG/B from LEIRIOS (now Smartesting) [1]. This tool uses B models [16] and is based on the symbolic animation of formal specifications with 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. Some of the most widely known MBT tools are substantially different to the TTF and Fastest [66, 67, 68, 69, 70, 71, 72, 73]. These differences range from the formal method the tools are based on—with some even relying on semi-formal notations like UML—to the specific technique they use to produce test specifications and cases. Most are based on some form of FSM and produce abstract test cases as sequences of events or transitions.

There are numerous papers proposing algorithms or techniques for automatically generating abstract test cases from Z specifications, but none of them show the implementation details of the proposed tools, nor are they based on the TTF or actually generate abstract test cases, but only test specifications. Therefore, it is unlikely to find MBT tools for the Z formal notation as automatic as Fastest. This situation, combined with the relative widespread use of the Z formal notation in industry and the advantages of the TTF, would make Fastest a good option for the Z community.

11. CONCLUSIONS AND FUTURE WORK

This paper presented Fastest, a tool that semi-automates the generation of unit tests from Z models based on the TTF. Simple and practical methods are proposed to prune unsatisfiable test specifications, to find abstract test cases from the surviving test specifications and to translate abstract test cases into natural language descriptions. Besides, an empirical analysis based on eleven models from different domains is shown, backing up the claims made on the efficiency and efficacy of the pruning and test case generation methods.

As the empirical study shows, it can be concluded that an apparently rough method to calculate abstract test cases proved to be reasonably efficient. In addition, the pruning method is an alternative to theorem proving, based on a library of contradictions that can be extended by users without previous knowledge of proving techniques. This method follows the seminal ideas of Dick and Faivre [19], in that they proposed to have an extensible set of elimination rules. This method has been more useful to the authors than a theorem prover in some real-world case studies.

The method for translating abstract test cases into natural language is a matter for further research. The solution presented above was successful in generating adequate NL descriptions of the test cases in one particular project. However, it has some limitations—for instance, it requires defining a new template for each operation; a task of still considerable size for large systems—that would not generalise well to specifications in other domains. Besides, Z specifications contain all the information necessary to automatically produce the templates for the operations in a system, regardless of its application domain. In particular, so called *designations* [74] might be of great help in generating NLTCTs. A thorough empirical evaluation of this method is also due.

Fastest's future is open in many directions. The most important one is perhaps to be able to deal with very large system specifications, i.e. specifications of entire systems not necessarily written as a combination of unit specifications. Some works has been started in such direction, together with Anthony Hall and Thomas Wilson from Altran Praxis (UK). This involves covering the entire Z notation, in particular schema types and their associated theory. Although the TTF has been extended to deal with axiomatic definitions and first-order quantifiers—not shown in this paper—some more research need to be done. A medium-term goal is to study the chances to integrate Fastest with a SMT solver with the intention to improve the generation of abstract test cases. Finally, the authors are currently working on the other steps of Figure 1, in particular abstract test case refinement and execution.

DOI: 10.1002/stvr

ACKNOWLEDGEMENTS

We would like to thank Flowgate Consulting and CIFASIS for their support in writing this paper. Richard Power from the NLG group at The Open University helped in finding financial support; and Eva Banik from the NLG Group at The Open University provided helpful comments on earlier versions of 7. Valdivino Santiago and N.L. Vijaykumar from Instituto Nacional de Pesquisas Espaciais (INPE), Miriam Alves from Instituto de Aeronáutica e Espaço (IAE), all of them from Brazil, and Gustavo Pérez from INVAP (Argentina), provided us excellent material for the case studies. We will be in debt with them for a long time. REVVIS funded the visits between INPE, IAE and CIFASIS through a CYTED grant. We want to thank Matthias Weber for providing us the steam boiler control software Z specification.

This paper is dedicated to the memory of David Carrington, one of the creators of the Test Template Framework, who passed away on 7 January 2011 after a long battle with cancer.

REFERENCES

1. Utting M, Legeard B. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 2006.
2. Hierons RM, Bogdanov K, Bowen JP, Cleaveland R, Derrick J, Dick J, Gheorghe M, Harman M, Kapoor K, Krause P, *et al.*. Using formal specifications to support testing. *ACM Comput. Surv.* 2009; **41**(2):1–76.
3. Stocks P, Carrington D. A Framework for Specification-Based Testing. *IEEE Transactions on Software Engineering* Nov 1996; **22**(11):777–793.
4. Grieskamp W, Gurevich Y, Schulte W, Veanes M. Generating finite state machines from abstract state machines. *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, ACM: New York, NY, USA, 2002; 112–122.
5. Legeard B, Peureux F, Utting M. A Comparison of the BTT and TTF Test-Generation Methods. *ZB '02: Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B*, Springer-Verlag: London, UK, 2002; 309–329.
6. Bernot G, Gaudel MC, Marre B. Software testing based on formal specifications: a theory and a tool. *Softw. Eng. J.* 1991; **6**(6):387–405.
7. Stocks P. Applying formal methods to software testing. PhD Thesis, Department of Computer Science, University of Queensland 1993.
8. MacColl I, Carrington D. Extending the Test Template Framework. *Proceedings of the Third Northern Formal Methods Workshop*, Ilkely, UK, 1998.
9. ISO. Information Technology – Z Formal Specification Notation – Syntax, Type System and Semantics. *Technical Report ISO/IEC 13568*, International Organization for Standardization 2002.
10. Cristiá M, Rodríguez Monetti P. Implementing and applying the Stocks-Carrington framework for model-based testing. *ICFEM*, *Lecture Notes in Computer Science*, vol. 5885, Breitman K, Cavalcanti A (eds.), Springer, 2009; 167–185.
11. Cristiá M, Albertengo P, Rodríguez Monetti P. Pruning testing trees in the Test Template Framework by detecting mathematical contradictions. *SEFM*, Fiadeiro JL, Gnesi S (eds.), IEEE Computer Society, 2010; 268–277.
12. Cristiá M, Plüss B. Generating natural language descriptions of Z test cases. *INLG*, Kelleher JD, Namee BM, van der Sluis I, Belz A, Gatt A, Koller A (eds.), The Association for Computer Linguistics, 2010.
13. Reiter E, Dale R. *Building Natural Language Generation Systems*. Cambridge University Press: Cambridge, UK, 2000.
14. Saaltink M. *The Z/EVES 2.0 User's Guide*. Ora Canada 1999.
15. Lamport L. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 2002.
16. Abrial JR. *The B-book: Assigning Programs to Meanings*. Cambridge University Press: New York, NY, USA, 1996.
17. Souza S, Maldonado J, Fabbri S, Masiero P. Statecharts Specifications: A Family of Coverage Testing Criteria. *CLEI'2000 - XXVI Latin-American Conference of Informatics*, CLEI, 2000.
18. 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* 2004; **34**(10):915–948.
19. Dick J, Faivre A. Automating the Generation and Sequencing of Test Cases from Model-Based Specifications. *FME '93: Proceedings of the First International Symposium of Formal Methods Europe on Industrial-Strength Formal Methods*, Springer-Verlag: London, UK, 1993; 268–284.
20. Cristiá M, Hollmann D, Albertengo P, Frydman CS, Monetti PR. A language for test case refinement in the Test Template Framework. *ICFEM*, *Lecture Notes in Computer Science*, vol. 6991, Qin S, Qiu Z (eds.), Springer, 2011; 601–616.
21. Spivey JM. *The Z notation: a reference manual*. Prentice Hall International (UK) Ltd.: Hertfordshire, UK, UK, 1992.
22. Maximiliano Cristiá. Fastest. URL `http://www.fceia.unr.edu.ar/~mcristia`, last access: November 2011.
23. Cristiá M, Rodríguez Monetti P, Albertengo P. *The Fastest 1.3.6 User's Guide*. CIFASIS 2010. URL `http://www.fceia.unr.edu.ar/~mcristia`.
24. The Eclipse Foundation. Eclipse. URL `http://www.eclipse.org/`, last access: November 2011.
25. CZT. CZT Eclipse Plugin. URL `http://www.cs.waikato.ac.nz/~marku/czt/eclipse.html`, last access: November 2011.
26. Hupponen T, Karlsson K, Laitinen J, Ojala O, Pirinen A, Seuranen E, Takkinen L. TeXlipse. URL `http://texlipse.sourceforge.net/`, last access: November 2011.

DOI: 10.1002/stvr

27. Malik P, Utting M. CZT: A framework for Z tools. *ZB*, *Lecture Notes in Computer Science*, vol. 3455, Treharne H, King S, Henson MC, Schneider SA (eds.), Springer, 2005; 65–84.
28. Freitas L, Utting M, Malik P, Miller T. Community Z Tools (CZT) project. URL http://czt.sourceforge.net, last access: November 2011.
29. Clements P, Garlan D, Bass L, Stafford J, Nord R, Ivers J, Little R. *Documenting Software Architectures: Views and Beyond*. Pearson Education, 2002.
30. Fitting M. *First-order logic and automated theorem proving (2nd ed.)*. Springer-Verlag New York, Inc.: Secaucus, NJ, USA, 1996.
31. Cristiá M, Albertengo P, Frydman C, Plüss B, Rodríguez Monetti P. Applying the Test Template Framework to aerospace software. *Proceedings of the 34th IEEE Annual Software Engineering Workshop*, IEEE Computer Society: Limerik, Irland, 2011. — to be published.
32. Page A, Johnston K, Rollison B. *How We Test Software at Microsoft*. Microsoft Press, 2008.
33. Kernighan BW, Ritchie DM. *The C Programming Language Second Edition*. Prentice-Hall, Inc., 1988.
34. Saaltink M. The Z/EVES System. *ZUM '97: The Z Formal Specification Notation*, Bowen J, Hinchey M, Till D (eds.), 1997; 72–85.
35. Saaltink M. The Z/EVES mathematical toolkit version 2.2 for Z/EVES version 1.5. *Technical Report*, ORA Canada 1997.
36. Coq Development Team. *The Coq Proof Assistant Reference Manual, Version 8.2*. LogiCal Project 2008.
37. Gomes CP, Kautz H, Sabharwal A, Selman B. Satisfiability solvers. *Handbook of Knowledge Representation*, *Foundations of Artificial Intelligence*, vol. 3. Elsevier, 2008; 89–134.
38. Nieuwenhuis R, Oliveras A, Tinelli C. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM* November 2006; **53**:937–977, doi:http://doi.acm.org/10.1145/1217856.1217859. URL http://doi.acm.org/10.1145/1217856.1217859.
39. ECSS. Space Engineering – Ground Systems and Operations: Telemetry and Telecommand Packet Utilization. *Technical Report ECSS-E-70-41A*, European Space Agency 2003.
40. Van Deemter K, Krahmer E, Theune M. Real versus template-based Natural Language Generation: a false opposition? *Computational Linguistics* 2005; **31**(1):15–24.
41. Bishop M. *Computer Security. Art and Science*. Addison Wesley, 2003.
42. Hörcher HM, Peleska J. Using Formal Specifications to Support Software Testing. *Software Quality Journal* 1995; **4**:309–327.
43. Helke S, Neustupny T, Santen T. Automating Test Case Generation from Z Specifications with Isabelle. *Lecture Notes in Computer Science*, Springer-Verlag, 1997; 52–71.
44. Kolyang, Santen T, Wolff B. A structure preserving encoding of Z in Isabelle/HOL. *TPHOLs '96: Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics*, Springer-Verlag: London, UK, 1996; 283–298.
45. Büssow R, Weber M. A steam-boiler control specification with Statecharts and Z. *Formal Methods for Industrial Applications, Specifying and Programming the Steam Boiler Control*, Springer-Verlag: London, UK, 1996; 109–128.
46. Utting M. Commercial MBT Tools. URL http://www.cs.waikato.ac.nz/research/mbt/, last access: November 2010.
47. Murray L, Carrington D, Maccoll I, Strooper P. TinMan - A Test Derivation and Management Tool for Specification-Based Class Testing. *In Technology of ObjectOriented Languages and Systems (TOOLS)*, 1999; 222–233.
48. Murray L. Software Requirements Specification for TinMan - Version 1.0. *Technical Report 99-02*, The Univesity of Qeensland 1999.
49. Ammann P, Offutt J. Using formal methods to derive test frames in category-partition testing. *Compass'94: 9th Annual Conference on Computer Assurance*, National Institute of Standards and Technology: Gaithersburg, MD, 1994; 69–80. URL citeseer.ist.psu.edu/ammann94using.html.
50. Hall PAV. Towards testing with respect to formal specification. *Proc. Second IEE/BCS Conference on Software Engineering*, no. 290 in Conference Publication, IEE/BCS, 1988; 159–163.
51. Hierons RM, Sadeghipour S, Singh H. Testing a system specified using Statecharts and Z. *Information and Software Technology* February 2001; **43**(2):137–149, doi:10.1016/S0950-5849(00)00145-2. URL http://dx.doi.org/10.1016/S0950-5849(00)00145-2.
52. Hierons RM. Testing from a Z specification. *Software Testing, Verification & Reliability* 1997; **7**:19–33, doi:10.1002/(SICI)1099-1689(199703)7:1⟨19::AID-STVR124⟩3.0.CO;2-N.
53. Burton S. Automated Testing from Z Specifications. *Technical Report*, Department of Computer Science – University of York 2000.
54. Bouquet F, Legeard B, Peureux F. Constraint logic programming with sets for animation of B formal specifications. *CL'00 Workshop on (Constraint) Logic Programming and Software Engineering (LPSE'00)*, London, UK, 2000.
55. Jackson D. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.
56. Bjørner N, de Moura L. $Z3^{10}$: Applications, enablers, challenges and directions. *Sixth International Workshop on Constraints in Formal Verification*, 2009.
57. Dutertre B, de Moura L. System description: Yices 1.0. *Proceedings of the 2nd SMT competition, SMT-COMP'06*, Seattle, USA, 2006.
58. Barrett C, Tinelli C. CVC3. *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)*, *Lecture Notes in Computer Science*, vol. 4590, Damm W, Hermanns H (eds.), Springer-Verlag, 2007; 298–302. Berlin, Germany.
59. Boyapati C, Khurshid S, Marinov D. Korat: Automated testing based on Java predicates. *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, ACM Press, 2002; 123–133.
60. Hèam PC, Nicaud C. Seed: an easy-to-use random generator of recursive data structures for testing. *IEEE International Conference on Software Testing, Verification and Validation (ICST 2011)*, IEEE Computer Society,

2011.
61. Punshon JM, Tremblay JP, Sorenson PG, Findeisen PS. From formal specifications to natural language: a case study. *Proceedings of the 12th international conference on Automated software engineering (formerly: KBSE)*, ASE '97, IEEE Computer Society: Washington, DC, USA, 1997; 309–.
62. Salek A, Sorenson P, Tremblay J, Punshon J. The REVIEW system: From formal specifications to natural language. *Proceedings of the First International Conference on Requirements Engineering*, 1994; 220–229.
63. Coscoy Y. A natural language explanation for formal proofs. *Selected papers from the First International Conference on Logical Aspects of Computational Linguistics*, Springer-Verlag: London, UK, 1997; 149–167.
64. Bertani A, Castelnovo W, Ciapessoni E, Mauri G. Natural language translations of formal specifications for complex industrial systems. *Atti del sesto congresso dell'Associazione Italiana per l'Intelligenza Artificiale*, Pitagora Editrice: Bologna, Italy, 1999; 185–194.
65. Lavoie B, Rambow O, Reiter E. Customizable descriptions of object-oriented models. *Proceedings of the fifth conference on Applied natural language processing*, Association for Computational Linguistics: Morristown, NJ, USA, 1997; 253–256.
66. Conformiq. URL http://www.conformiq.com, last access: November 2011.
67. Smartesting. URL http://www.smartesting.com, lass access: November 2011.
68. Rational Statemate. URL http://www.ibm.com/developerworks/rational/, last access: November 2011.
69. Model JUnit. URL http://www.cs.waikato.ac.nz/~marku/mbt/modeljunit/, last access: November 2010.
70. Requirements-Based Automated Verification. URL http://www.t-vec.com/solutions/rave.php, last access: November 2011.
71. T-vec Tester for Simulink and Stateflow. URL http://www.t-vec.com/solutions/simulink.php, last access: November 2011.
72. Reactis. URL http://www.reactive-systems.com, last access: November 2011.
73. Veanes M, Campbell C, Grieskamp W, Schulte W, Tillmann N, Nachmanson L. Model-based testing of object-oriented reactive systems with Spec Explorer. *Formal methods and testing*, Hierons RM, Bowen JP, Harman M (eds.). Springer-Verlag: Berlin, Heidelberg, 2008; 39–76.
74. Jackson M. *Software requirements & specifications: a lexicon of practice, principles and prejudices*. ACM Press/Addison-Wesley Publishing Co.: New York, NY, USA, 1995.

DOI: 10.1002/stvr