Pruning Testing Trees in the Test Template Framework by Detecting Mathematical Contradictions

Maximiliano Cristiá
Flowgate Consulting and CIFASIS
Rosario – Argentina
mcristia@flowgate.net

Pablo Albertengo
Flowgate Consulting
Rosario – Argentina
palbertengo@flowgate.net

Pablo Rodríguez Monetti
Flowgate Consulting and UNR
Rosario – Argentina
prodriguez@flowgate.net

***Abstract*—Fastest is an automatic implementation of Phil Stocks and David Carrington's Test Template Framework (TTF) [1], a model-based testing (MBT) framework for the Z formal notation. In this paper we present a new feature of Fastest that helps TTF users to eliminate inconsistent test classes automatically. The method is very simple and practical, and makes use of the peculiarities of the TTF. Perhaps its most interesting features are extensibility and ease of use, since it does not assume previous knowledge on theorem proving. Also we compare the solution with a first attempt using the Z/EVES proof assistant and with the HOL-Z environment. At the end, we show the results of an empirical assessment based on applying Fastest to four real-world, industrial-strength case studies and to six toy examples.**

I. Introduction

In [2] we presented Fastest[1], the first automatic tool that partially implements the Test Template Framework (TTF). The TTF is a framework for model-based testing (MBT) specially well suited for unit testing from Z specifications proposed by Phil Stocks and David Carrington in [1] [3] [4]. Within the TTF the input space of a Z operation is partitioned into so called *test classes*[2] which form a testing tree as shown in Fig. 2. Stocks and Carrington suggest that test cases should be derived only from the leaves of such trees. However, 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 term. Then, these leaves should be pruned from the testing tree prior to start the process of deriving test cases. Hence, the problem of pruning testing trees is the problem of determining either those test classes that are satisfiable or those test classes that are unsatisfiable. The first approach implies to build a SMT-like tool but implementing the whole Z mathematical toolkit [5]—certainly not a trivial task given the number of different theories defined in the toolkit. Precisely, the rich mathematical language used by Z specifiers makes the problem of pruning testing trees harder because most of the contradictions are due to mathematics and not to logic. In this paper, on the other hand, we present an implementation of the second approach which is closer to automatic theorem proving, although it is far more simple.

The problem of pruning inconsistent test classes is as important as the problem of finding test cases from satisfiable test classes. This is so because of two reasons. First, the TTF tends to yield a large number of unsatisfiable test classes. Second, since the problem of determining the set of satisfiable test classes is undecidable, any automatic method for finding test cases will yield an incomplete answer. If such a method could not find a test case for a given test class, is it because the test class is unsatisfiable or is it because the method is (necessarily) incomplete? Likewise, any automatic method for pruning inconsistent test classes will also be incomplete—because of undecidability, too. Precisely, these are the reasons for which a tool like Fastest must implement both methods: they will complement each other, leaving to manual inspection just a small fraction of all test classes. In [2] we presented a simple but practical method for finding test cases from test classes—on average, our method can find a test case for 90% of the *satisfiable* test classes. Similarly, we devised the method introduced in this paper with essentially one goal in mind: to dramatically reduce the time needed to prune unsatisfiable test classes from testing trees produced by the TTF.

The paper is structured as follows. The next section presents a brief introduction to the TTF. Section III is the core of the paper where we introduce our method. We analyse how to certify a key aspect our method in Sect. IV. Section V presents the results of an empirical assessment of the implementation. Our work is compared with similar approaches in Sect. VI, particularly against using theorem provers. Finally, Sect. VII describes our conclusions and future work.

II. The Test Template Framework

In this section we introduce the TTF by means of an example, without mention any particular implementation or tool—for a deeper introduction see [2] or the original work [1]. We assume the reader is fluent in the Z notation [6].

[1]Fastest is publicly available at `http://www.flowgate.net` in the Tools section.

[2]Also called *test objectives*, *test specifications*, *test templates*, etc.

$[SENSOR]$

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

KMROk

$\Delta MaxReadings$

$s? : SENSOR;\ r? : \mathbb{Z}$

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

$smax\ s? < r?$

$smax' = smax \oplus \{s? \mapsto r?\}$

*KMRE*2

$\Xi MaxReadings$

$s? : SENSOR$

$r? : \mathbb{Z}$

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

$r? \leq smax\ s?$

$KMRE1 ==$

$[\Xi MaxReadings;\ s? : SENSOR \mid s? \notin \operatorname{dom} smax]$

$KMR == KMROk \lor KMRE1 \lor KMRE2$

Figure 1: A Z model for a simple pool of sensors.

A. A Simple Pool of Sensors

The Z model of Fig. 1 describes a simple pool of sensors which records the highest reading of each sensor. The *KMR* operation[3] takes as input a sensor ID, $s?$, and a reading, $r?$, of it. If $s?$ is a valid ID and if $r?$ is greater than the current reading of $s?$, *KMR* replaces the current reading with $r?$. If some condition does not hold, then the operation fails and nothing is changed.

B. Testing Tactics, Test Classes and Testing Trees

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

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

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

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

Stocks and Carrington suggest to divide the *VIS* into equivalence classes, called *test classes*, by applying one or more *testing tactics*[4]. The test classes obtained in this way can be further subdivided into more test classes by applying other testing tactics. This procedure continues until the engineer is satisfied with the test coverage. Within the TTF all these test classes are represented as a *testing tree*, as shown in Fig. 2. Test cases are taken only from the leaves of the testing tree.

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

[4] Also called testing strategies.

The authors of the TTF defined a number of testing tactics that together provide a sound method for calculating tests objectives. Furthermore, they propose that new tactics should be added for particular projects, systems, requirements, etc. We will apply two testing tactics to *KMR* described in [1]. First we apply Disjunctive Normal Form (DNF) to the *VIS* and then Standard Partitions is applied to the expression $smax\ s? < r?$ of *KMROk*. The resulting testing tree is shown in Fig. 2. Some of the nodes of the testing tree are shown in Fig. 3 as Z schema boxes. We chose to include in the figure only unsatisfiable classes since this is the main issue of this article.

The second step of the TTF methodology suggests to prune the unsatisfiable test classes from the testing tree, because it is impossible to find test cases from them. For instance, classes *KMR_SP*_9 and *KMR_SP*_14 must be pruned, among others—see Fig. 3. The predicate of *KMR_SP*_9 is unsatisfiable because $smax\ s?$ is undefined since $s? \notin \operatorname{dom} smax$, while *KMR_SP*_14 is inconsistent because its predicate is an arithmetic contradiction.

Stocks and Carrington do not give any recipe on how pruning can be automated within the TTF. Such a method is the core of this article and, as far as we know, it is the first proposal specifically tailored to the TTF, although it might be interesting for other approaches as well.

The TTF follows with finding at least one element for each of the remaining leaves of the testing tree. This step is explained in [2] and [3].

III. Detecting Mathematical Contradictions

Fastest applies DNF by default thus making all the test class predicates to be conjunctions of atomic predicates. The other testing tactics implemented in Fastest conjoin more atomic predicates to those already appearing in the test classes, as prescribed by the TTF. Hence, the leaves of any testing tree are conjunctions of atomic predicates. On the other hand, we have said that it is impossible to find an abstract test case out of a test class when: (a) the class predicate is a contradiction, or (b) there is some undefined term within the predicate (as in class *KMR_SP*_6 shown in Fig. 3). Our method treats contradictions and undefinition in the same way, so in the rest of the paper we will simplify the exposition talking only about contradictions.

Since the predicate of a leaf in a testing tree is a conjunction of atomic predicates, then the predicate is a contradiction if and only if: (i) there exist atomic predicates p and $\neg\ p$ in it, or (ii) there is some mathematical contradiction between two or more atomic predicates. The first kind is easy to deal with, the second kind is the core of the problem.

Let us analyze the other unsatisfiable test classes shown in Fig. 3, in order to have an idea of what are typical contradictions. *SCAddCat_SP*_1 is unsatisfiable because the proposition $\{c?\} = \{\}$ is false; in *SCAddCat_SP*_5 the

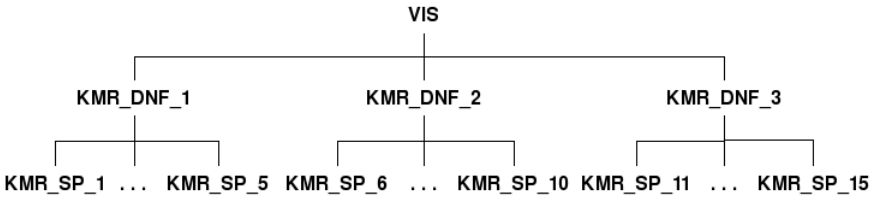

Figure 2: Testing tree of *KMR*.

KMR_SP_9

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

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

SCAddCat_SP_1

$$level : \mathbb{Z}$$
$$categs : \mathbb{P}\, CATEGORY$$
$$c? : CATEGORY$$

$$c? \notin categs \land categs = \{\}$$
$$\{c?\} = \{\}$$

Del_SP_1

$$procs : PID \pfun PSTATUS$$
$$p? : PID$$

$$p? \in \operatorname{dom}(procs \rhd \{waiting\})$$
$$procs = \{\}$$

KMR_SP_14

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

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

SCAddCat_SP_5

$$level : \mathbb{Z}$$
$$categs : \mathbb{P}\, CATEGORY$$
$$c? : CATEGORY$$

$$c? \notin categs$$
$$\{c?\} \neq \{\}$$
$$categs \neq \{\} \land categs \subset \{c?\}$$

Deposit_SP_18

$$sa : NSA \pfun BALANCE$$
$$n? : NSA$$
$$a? : \mathbb{Z}$$

$$a? \leq 0$$
$$sa = \{\}$$
$$\{n? \mapsto (sa\ n? + a?)\} \neq \{\}$$

Figure 3: Z schema boxes representing unsatisfiable test classes of the *KMR* example and others.

conjunction $categs \neq \{\} \land categs \subset \{c?\}$ is a contradiction; in *Del_SP_1* there is another mathematical contradiction since $p?$ cannot belong to $\operatorname{dom}(procs \rhd \{waiting\})$ when *procs* is an empty set; and in *Deposit_SP_18* the application $sa\ a?$ is undefined because *sa* is empty—note that here the undefinition comes from a different atomic predicate than in *KMR_SP_9*, also in Fig. 3.

As the reader can see, some of these contradictions depend on the particular semantics of the Z mathematical toolkit, 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. This situation implies that 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. Indeed, our first attempt was to use the Z/EVES theorem prover [7], but it proved to be less effective and efficient than the method implemented in Fastest as we show in Sect. VI.

Hence, Fastest provides a command named `prunett` that analyzes the predicate of each leaf in a testing tree to determine if the predicate is a contradiction or not. Since this problem is undecidable, `prunett` implements a best-effort algorithm that can be improved by users. 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, the following two elimination theorems are included in the library:

ETheorem SingletonIsNotEmpty $[x : X]$

$$\{x\} = \{\}$$

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

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

Note that the contradiction in *SCAddCat_SP_1* is an instance of SingletonIsNotEmpty, while NotSubsetOfSingleton generalizes the contradiction present in *SCAddCat_SP_5*.

The pseudocode of the pruning algorithm implemented by Fastest is described in Fig. 4. We illustrate the pseudocode in Section III-A because first we need to introduce some key design concepts about the method, as follows.

LaTeX: Elimination theorems in the library are written in LaTeX using the CZT package [8], which conforms to the ISO Standard of the Z notation [9].

Initialization Stage

1) Check the elimination theorems and load them.
2) Combine equivalence rules with the atomic predicates of every elimination theorem in the library.
3) Convert each elimination theorem into a regular expression.

When `prunett` is executed

1) Prune test classes with predicates of the form $\cdots \wedge p \wedge \cdots \wedge \neg p \wedge \cdots$.
2) For each leaf predicate P in the testing tree:
   a) Convert P into a string.
   b) For each elimination theorem T in the library with formal parameters $p_1 : T_1, \ldots, p_n : T_n$:
      i) If P's atomic predicates pattern-match the atomic predicates of T, then:
         A) If the expressions of P that pattern-matched the formal parameters of T type-check against $T_1, \ldots, T_n$ or their subtypes, then prune P and start the next iteration in 2.

      End for each.

   End for each.

Figure 4: `prunett` can be run after testing trees have been generated.

Formal Parameters: 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 different literals. 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 Body of Elimination Theorems: 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 supported by Fastest, the names of the formal parameters or the reserved words somewhere, anything and eval, which are explained below.

Somewhere: somewhere takes a parameter consisting of a Z LaTeX string. For instance, the library contains the following elimination theorem:

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

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

somewhere(*string*) is rather similar to the regular expression $*string*$—for a full description of the semantics see [10]. When the algorithm finds such a directive it tries to match the regular expression in any of the atomic predicates of a test class' predicate.

Anything: anything 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 $[s : \operatorname{seq} X;\ n : \mathbb{N}]$

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

Evaluations: eval 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$(N \leq M)$
$n + N \,..\, (n + M) = \{\}$

This sentence evaluates the Boolean expression; if it is *true* and all of the other conjuncts of the theorem are found in the test class, then the test class is pruned. If the Boolean expression evaluates to *false*, the test class is not pruned.

Equivalence Rules: Fastest applies equivalence rules, taken from the library, to the elimination theorems whenever possible. Besides, it applies by default the rules listed in Table 5a. This implies, for instance, that the engineer does not need to write the following theorem:

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

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

because the library already contains the following one:

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.

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
$X \leftrightarrow Y$	$X \leftrightarrow Y$, $X \pfun Y$, $X \fun Y$, $\operatorname{seq} Y$
$X \pfun Y$	$X \pfun Y$, $X \fun Y$, $\operatorname{seq} Y$
$\mathbb{Z}$	$\mathbb{Z}$, $\mathbb{N}$

(b) Subtyping rules

Figure 5: Fastest applies by default these equivalence and subtyping rules.

Subtyping Rules: Fastest also applies some simple subtyping rules when 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[5]. For instance $X \fun Y$ is a subtype of $X \pfun Y$ which in turn is a subtype of $X \leftrightarrow Y$. The subtyping rules applied by Fastest are listed in Table 5b.

A. The Algorithm

In this section we will comment the pseudocode listed in Fig. 4. The algorithm is implemented in Java and is based essentially on regular expressions, pattern matching and string search. Basically, each elimination theorem is converted from a set of Java objects into a regular expression and the predicate of each test class is converted into a string[6]. Then, we simply try to match the string against the regular expression. Regular expressions become rather complex because they include back references to capture the actual parameters that match in a test class' predicate [11], alternatives to capture equivalence rules, etc.

Next, we explain in more detail step 2 of the initialization stage and step 2(b)iA of `prunett`. During initialization, Fastest loads the elimination theorem library, parses it, builds an AST, and checks some consistency issues of each elimination theorem, all this is performed with the tools provided by the CZT project [8].

In step 2 of the initialization stage equivalent rules are combined with the elimination theorems as alternatives of regular expressions. As we have said, equivalent rules are lists of atomic predicates. If $e_1, \ldots, e_n$ is an equivalent rule and there is an atomic predicate p in an elimination theorem which happens to be a pattern for some e_i, then the algorithm replaces p by $e_1 \mid \cdots \mid e_n$—where $\mid$ is the "or" operator for regular expressions—conveniently changing the formal parameters in all the e_i for the formal parameters of p. For instance, if the equivalence rule is $R = \{\} \Leftrightarrow \operatorname{dom} R = \{\}$ and the atomic predicate is $R \oplus G = \{\}$, then the atomic predicate becomes $R \oplus G = \{\} \mid \operatorname{dom}(R \oplus G) = \{\}$.

Once the library was successfully loaded and there is at least one testing tree, the user can run `prunett`. If the string form of a test class' predicate matches against the regular expression form of an elimination theorem, then in step 2(b)iA we type-check the substrings of the former that matched against the formal parameters of the latter—this matching is implemented with back references [11]. If this last check passes then the test class is pruned. The substrings that matched against the formal parameters are converted back onto objects of the AST. Then, this objects and the object-oriented form of the elimination theorem are used to perform the type-checking by means of the tools provided by CZT. This type-checking includes the subtyping rules mentioned above, as follows. If an elimination theorem has a parameter of type U then, any expression whose type is a subtype of U is considered to type-check against U. Hence, if there is a test class containing a term f of, say, type $\mathbb{N} \pfun CHAR$, then the parameter R of the elimination theorem NotInEmptyDom shown above will be substituted by f because $X \pfun Y$ is a subtype of $X \leftrightarrow Y$, $\mathbb{N}$ matches X and $CHAR$ matches Y.

B. Distributed Pruning

Fastest is a distributed system [2]. The user can configure the tool to distribute some tasks. When Fastest runs on a single computer we say it runs in *application mode* and when it runs on more than one we say it is in *distributed mode*. When running in distributed mode, `prunett` sends test classes to different testing servers so they can prune them in parallel.

At the beginning of our research we thought distributed pruning would be faster than assigning the task just to the client computer. However, the algorithm proved to be so fast in application mode (see Sect. V) that we need to further experiment to see when distribution is actually an aid or not, because of the time penalization incurred during network transmission, synchronization, etc.

C. Discussion of the Method

After implementing the basic features of Fastest we applied it to some of the case studies listed in Table I at page 8 [2]. We quickly discovered three facts: (a) TTF's testing trees tend to have a large number of unsatisfiable test classes; (b) all of them correspond to a few dozens of trivial mathematical contradictions; and (c) new projects or new

[5]It should be noted that the notions of type and subtype in Z are not as strong as in other typed formalisms or tools such as Coq or PVS. For this reason we do not give a more precise definition of subtyping.

[6]Actually, both regular expressions and strings are also Java objects but of very different types.

testing tactics might produce new kinds of contradictions. Initially, we considered SMT solvers but it was impossible to find one implementing the Z mathematical toolkit [5]. Our first approach was, then, to use a general theorem prover as suggested in [12]. So, we used Z/EVES to prune testing trees as is described in Sect. VI-A. It worked reasonably well but it did not prune all the unsatisfiable test classes and it took too long in some experiments (see Table II at page 9). Besides, it was not clear how users would extend the method without learning theorem proving. Then, we thought it would be interesting to try out something specifically tailored to the TTF.

From the very beginning we knew that a complete solution was impossible. Hence, we devised a method that should *work in practice*, although it might not be sophisticated nor elegant. For us, to "work in practice" means that at least 80% of the unsatisfiable test classes 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, we are dealing with testing. Furthermore, if the method could be improved as new projects were executed, then the "work in practice" criteria might be reached as times goes and more users work with the tool.

Our empirical results confirm that the method presented in this paper meets, and perhaps exceeds, our "work in practice" criteria—see Sect. V. From the method description it is easy to see that the more elimination theorems in the library, the more test classes will be pruned by `prunett` without user intervention. However, what if it worked just for the models with which we experimented? Is it sufficiently general? Since the method is based on maintaining a library of elimination theorems, how general are them? How many contradictions can they represent? We think the answer is in the fact that the language to write elimination theorems is essentially the same than the language in which the contradictions that have to be detected are written: i.e. the Z notation. Then, if elimination theorems and contradictions are both expressed in the same language, there is no reason to believe that some contradiction cannot be detected. In the worst case, users can add elimination theorems that are exactly the same as the contradictions they are seeing in test classes. For instance, if the contradiction in a test class is $x = 73 \land x = 12$, then the corresponding elimination theorem might be:

ETheorem 73neq12 $[x : \mathbb{Z}]$
$x = 73$
$x = 12$

However, writing the same number of elimination theorems than the contradictions appearing in every project, is as impractical as inspecting test classes by hand. As Table I at page 8 shows, the method pruned more than 2,000 unsatisfiable test classes with only 52 elimination theorems. This was possible because the language allows users to write highly parametrized elimination theorems, making them general patterns of mathematical contradictions. This generality is achieved through: the name and type of parameters, subtyping and equivalence rules and the somewhere, anything and eval directives. In this way 73neq12 became ExcludedMiddle shown at page 8.

Furthermore, due to the peculiarities of the TTF many elimination theorems can be predicted and, thus, can be included in the library before starting a new project. In effect, since testers known in advance the list of available testing tactics they can foresee some contradictions as the result of applying them. For instance, if the Standard Partition defined in [1] for the set union operator, $\cup$, is going to be applied, then it is easy to predict that there will be test classes of the form $x \in A \land A = \varnothing$, where the first atomic predicate is part of the specification of a given operation and the second is added as the result of applying the tactic. Therefore, testers might add elimination theorems before using Fastest so they have a certain warranty that many test classes will be pruned without their intervention, even in the first project.

Finally, we want to make it clear that the method does not make any kind of deduction as theorem proves 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'. Of course, that reasoning must take into consideration equivalence and subtyping rules and the other generalization mechanisms.

IV. Certified Elimination Theorems

The user has the responsibility of maintaining the elimination theorem library. However, this possibility involves a risk: the user might add an elimination theorem which does not actually represent a contradiction. Adding an invalid elimination theorem might result in pruning satisfiable test classes, which in turns means less test cases.

To avoid this risk, users should (formally) prove that what they think is a contradiction, really is. This process is sometimes called *certification* because it involves a proof assistant. We have used Z/EVES to certify 50 of the 52 elimination theorems of Fastest's library. Those elimination theorems that were not certified make use of the reserved word somewhere which is not easy to represent in Z. We will further investigate how to certify them too.

Since the Z syntax expected by Z/EVES is different from the Z standard (the one used by Fastest), the presence of eval and other directives, and the fact that elimination theorems are the opposite of a theorem, we had to make some minor adjustments to the library in order to export it to Z/EVES. The changes were the following:

- Change the way in which formal parameters are written.
- Negate the predicate of the elimination theorem.

- Add the conditions of the eval sentences.
- Appropriately remove the anything directives.
- When const is used for two or more parameters, add a predicate stating that all these parameters are distinct from each other.

For instance, RangeNotEmpty (see page 4) must be translated into:

Theorem RangeNotEmpty
$$\forall n, N, M : \mathbb{N} \bullet \neg (N \neq M \land N \leq M \land n + N \mathinner{..} (n + M) = \{\})$$

After exporting the library we loaded the theorems in Z/EVES and made the 50 proofs[7]. 33 (66%) of the proofs required only the `prove` command (i.e. we can say those proofs were automatic), we had to introduce five lemmas to prove only 5 (10%) of the theorems because their proofs are rather complex, and 12 (24%) required more than one proof command but we made no use of the lemmas.

This way of certifying elimination theorems is not intended to be performed by end users. We included it in this paper for two reasons: (a) it shows another area where theorem provers and MBT can be combined; and (b) it shows that most elimination theorems are so simple that can be proved automatically. We think that any software engineer who is used to the Z formal notation could write other elimination theorems as they are needed. A full production version of Fastest could provide certification in two different ways: (a) as a service provided by the builder; and (b) as a seamless integration with a theorem prover.

V. Empirical Assessment

We devised this method with essentially one goal in mind: to reduce the time needed to prune unsatisfiable test classes from testing trees produced by the TTF. To reduce this time, automation, and consequently performance, were the keys. Since the problem is undecidable, full automation was impossible. Then, we thought in a method whose level of automation would grow from project to project as users improve it with more information. Ease of use or simplicity became, then, key success factors. This empirical assessment was made to measure whether the method meets our expectations or not. Hence, we run ten experiments to measure: (a) how much computing time `prunett` needs to prune testing trees; (b) how many elimination theorems need to be added from project to project in relation with the new mathematical theories used in new projects; and (c) how complex elimination theorems may be.

These parameters were measured by applying Fastest to the ten case studies shown in Table I. Six of them are toy examples borrowed from the literature or proposed by us, while the remaining four are real-world, industrial-strength problems. These case studies belong to eight different application domains. More detailed descriptions of the first eight are available in [2, page 179], the ninth is explained in Sec. VI-B and the last is similar to Plavis. In these case studies testing trees were built by applying two or more testing tactics—DNF and some other tactics since the former is applied by default. Table I gives both an idea of the complexity of the Z models and the results of each experiment. Columns are as follows[8]: **LOZC** stands for number of lines of Z code (in LaTeX format), **State** indicates the number of state variables; **Op** gives the number of Z operations involved in the experiment; **Classes** is the total number of leaves right after generating the testing trees; **Atomic** is the average number of atomic predicates present in the test classes (this average was calculated for all the operations of column **Op**); **U** is the number of unsatisfiable test classes (manual analysis); **Th** is the *cumulative* number of elimination theorems necessary to prune all the unsatisfiable test classes; **PC** is the number of *possible* contradictions found in average in each unsatisfiable test class; **Time** is the time needed to prune; **Theories** are the new mathematical theories used by each case study with respect to the previous ones.

Two columns deserve some comments. In **PC** we tried to detect how many contradictions are in each test class. Unfortunately, the algorithm is not prepared to provide *accurately* such information, because once it finds an elimination theorem that can prune a test class it stops searching. However, Fastest provides another command, called `searchtheorems`, which returns a list of the elimination theorems that might prune a given test class. Then, in column **PC** we report the average of results returned by that command. Therefore, column **PC** shows that, in average, there might be more than one contradiction in each test class. The Steam Boiler case study is particular in this regard because Fastest found only logical contradictions, i.e. no mathematical contradiction was found. As can be seen from the table, the number of possible contradictions tends to increase with the number of atomic predicates, as would be expected since test classes are built automatically from general testing tactics.

To better explain the meaning of the column labeled with **Th**, consider the following. Initially, we set up an elimination theorem library containing only the next three elimination theorems, which are related to logic and typing rather than to mathematics:

ETheorem NatDef $[n : \mathbb{N}]$
$$\neg\ 0 \leq n$$

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

[7]The theorems and proof scripts ready to be loaded into Z/EVES are available at `http://www.flowgate.net` in the Tools section.

[8]Some are self explanatory.

Case Study	R/T	LOZC	State	Op	Classes	Atomic	U	Th	PC	Time	Theories
Pool of Sensors	Toy	46	1	1	15	8.3	8	7	2.2	0.5s	Integer inequalities
Symbol Table	Toy	78	1	3	26	7.2	16	16	2.3	0.6s	Basic set theory
Lift	Toy	152	6	3	17	12.6	1	17	4	0.5s	
Security Class	Toy	172	4	7	36	9.2	16	17	2.6	0.8s	
Savings Accounts	Toy	171	1	5	97	8.4	75	20	3.7	3s	Relational domain
Scheduler	Toy	240	3	10	213	10.3	164	33	3.2	7s	Cardinality, singletons, relational range
Plavis	Real	608	13	13	232	13.1	50	38	3.7	15s	Sequences
SWPDC	Real	1,238	18	17	201	27.0	56	52	5	31s	Integer ranges
Steam Boiler	Real	591	12	1	400	7.5	336	52	lc	3s	
ECSS-E-70-41A	Real	774	13	5	1,226	18.8	856	52	5.8	2m18s	

Table I: Summary of the experiments. All the experiments were conducted over the same hardware and software platform: an Intel Centrino Duo of 1.66 GHz with 1 Gb of main memory, running Linux Ubuntu 8.04 with kernel 2.6.24-24-generic and Java SE Runtime Environment (build 1.6.0_14-b08). Fastest was run in application mode with the following command `java -Xss8M -Xms512m -Xmx512m -jar fastest.jar`.

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

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

$$x = y$$
$$x = z$$

Then, the simplest model (Pool of Sensors) was loaded in Fastest, testing tactics were applied, testing trees were generated, and finally `prunett` was executed—we call this the *script*. If `prunett` could not prune all the unsatisfiable test classes, then we extended the library with the minimum amount of elimination theorems to do that, and the script was run again. In other words, the three initial elimination theorems could not prune all of the unsatisfiable test classes of the first case study, so we added four new elimination theorems making a total of 7. At this point we measured the computing time needed to prune. This procedure was repeated for all the case studies in increasing order of complexity (**LOZC**), except for the last two which were executed at the end because they have considerable more leaves and add no new mathematical theories. For instance, in order to prune all the test classes of the second case study we needed to add 9 elimination theorems, yielding a total of 16 for the second experiment. Again, at this point we measured the pruning time for the experiment. Tables II and III, studied in Sect. VI, will give more insight on the performance and effectiveness of `prunett`.

In order to determine the ease of use or simplicity of the method we measured the complexity of elimination theorems. In the library used for conducting the experiments the largest elimination theorems have three atomic predicates, but most have two. This is no more complex than the theorems present in theorem provers' libraries. Furthermore, the atomic predicates appearing in the elimination theorems are simple generalizations of the contradictions found in test classes, i.e. the user does not need to do any kind of logical inference or deduction to be able to write them.

Although this empirical assessment might need a bigger sample set, we think it shows a good tendency that confirms our expectations. The computing times obtained so far are excellent compared to the time that manual inspection would take and with respect to similar approaches—see Sect. VI. The second measure—that the user intervention will decrease as new projects are executed—confirms that the method "works in practice". Precisely, Table I shows that every time a model not adding new mathematical theories is loaded into Fastest, almost no new elimination theorems are needed—see the rows were column **Theories** is empty and compare the values of column **Th** in those rows with the same values of their predecessors.

VI. Comparison with Similar Approaches

In this section we compare our work with a first attempt using the Z/EVES proof assistant, with the results reported in [12], and with other similar approaches. The key issues to compare are: the amount of inconsistent test classes that can be pruned automatically; the computing time required to prune; and the amount and simplicity of theorems needed to prune.

A. Pruning Testing Trees with Z/EVES

Before implementing the method described in this paper we used Z/EVES to prune testing trees. We wrote a `bash` script which takes the model and the test classes generated by Fastest and returns the list of inconsistent test classes found by Z/EVES. Firstly, the script translates the files into the LaTeX format used by Z/EVES[9]; secondly, it automatically generates a theorem of the form:

Theorem UNSAT_TestClass
$$\neg \text{ TestClass}$$

for each test class and adds the most powerful proof command, `prove by reduce`. Then, all this information is

[9]The translation works only for the case studies; Z/EVES does not accept the ISO standard for Z.

Case Study	Z/EVES		Fastest	
	Pruned	Time	Pruned	Time
Pool of Sensors	3	1s	8	0.5s
Symbol Table	11	2s	16	0.7s
Lift	1	11s	1	0.5s
Security Class	14	4s	16	0.8s
Savings Accounts	45	15s	75	3s
Scheduler	123	26s	160	7s
Plavis	19	6m50s	50	16s
SWPDC	21	16m31s	56	31s
Steam Boiler	159	23m9s	336	3s
ECSS-E-70-41A	728	1h20m51s	856	2m18s

Table II: Pruning with Z/EVES. Fastest was run with a library containing 52 elimination theorems, and both programs were run on the platform described in Table I.

Experiments	HOL-Z			Fastest		
	DNF	F	Time	DNF	F	Time
SB	48	14	1m41s	50	14	0.7s
SBW	8	3	5s	8	8	0.5s
SBW DNFs unfolded	42	6	1m36s	Impossible		
SBW direct	384	?	22h	400	64	3s

Table III: Comparison with HOL-Z. Helke and his colleagues run their experiments on a Sun Ultra-Sparc while we run them on the platform described in Table I.

loaded into Z/EVES. Finally, the script search the theorems that were proved.

Table II shows the results of running the script against the same case studies analyzed in Sect. V. It must be noted that the mathematical toolkit of Z/EVES contains 565 theorems and, so far, Fastest's has 52. As the reader can see, Fastest outperforms Z/EVES in all the key issues proposed for the comparison: it prunes more test classes and takes invariably less time than Z/EVES, with notably less theorems.

B. Test Class Simplification with Isabelle

In [12] the authors apply the Isabelle theorem prover to, among other things, eliminate unsatisfiable test classes. They use an encoding of Z in Isabelle, called HOL-Z [13], to MBT the *STEAM_BOILER_WAITING* (*SBW*) operation of the steam boiler control software specified in Z [14]. They recognize the need to eliminate test classes after applying DNF and show their results in terms of the number of simplified test classes and the computing time needed to do that for four experiments with *SBW*. In Table III we reproduce their results and ours after applying Fastest to the same experiments. The meaning of the columns is as follows: **DNF** is the number of test classes after applying DNF, **F** is the same number but after pruning (or simplifying) the DNF, and **Time** is the time needed to prune.

Due to space restrictions we cannot include the Z schema of *SBW*. We think that the reader should only know that *SBW* includes the state schema *SteamBoiler* (*SB*) which has a rather complex state invariant and it, in turn, includes some other state schema as well. In the first experiment the authors of [12] apply DNF to the state schema *SB* without unfolding the schema references appearing in it. In this way the main state invariant is written in DNF. In the second experiment they apply DNF to *SBW* without unfolding *SB* and the other schema references. In the third one, they "unfold the DNF of *SB* into the DNF of *SBW* yielding the test classes for this schema". This is impossible to do in Fastest because simplification is applied after the DNF of the outermost schema has been calculated. Experiment four "illustrates the use of exploiting the structure of the specification: unfolding *SB* in *SBW* and trying to compute the test classes in a single step".

As the reader can see from Table III, the computing times shown by Fastest are systematically much better that those obtained by Helke and his colleagues. Honestly, they performed the experiments on a Sun Ultra-Sparc while we did it on a much modern platform. However it is unlikely that the differences come solely from this, particularly in the fourth experiment.

On the other hand, our method yields the same number of test classes after pruning in the first experiment, although the numbers of test classes after DNF are different. The number of simplified test classes reported by Helke in the second experiment is strange because we were unable to reproduce it neither with Fastest nor with Z/EVES nor by hand. We believe that these differences might come from different versions of the steam boiler specification used by them and us—we obtained it from one of the authors of [14]. In the fourth experiment Helke does not inform the number of test classes after simplification. In this case the difference in the number of test classes after DNF comes from the difference in the same number of the first experiment, i.e. $384 = 48 \times 8$ and $400 = 50 \times 8$.

Considering just these experiments, we can conclude that Fastest is more efficient and at least as much effective than the HOL-Z environment in simplifying testing trees. Furthermore, it is not clear that HOL-Z can easily implement the TTF beyond calculating the DNF of Z operations. Maybe the better performance of Fastest steams from the fact that it was born as a specialized TTF-MBT tool for the Z notation, while HOL-Z is a Z proof environment mounted, in turn, on a general theorem prover.

C. Other Approaches

We could not find many other references dealing with the pruning problem in the context of model-based testing. In their seminal works, Stocks and Carrignton warn that false branches must be removed from testing trees [1]. Dick and Faivre [15] observe that contradictory sub-domains must be eliminated by applying an extensible set of rules. Most of the other references report some technique to simplify tests in some way but in quite different contexts. For example, perhaps the most recent and complete survey on formal

methods and testing [16] hardly cites the issue of pruning test classes. In [17] the authors use heuristics to reduce the search space of test sequences in abstract finite state machines. C. Meudec in his PhD thesis [18] discusses simplification for the VDM language. Doong and Frankl in [19][20] also simplify test sequences in the context of LOBAS algebraic specifications.

VII. Conclusions and Future Work

We have presented an alternative method to theorem proving for pruning unsatisfiable test classes from testing trees, based on codifying contradictions rather than tautologies. We believe this method follows the seminal ideas of Dick and Faivre [15], in that they proposed to have an extensible set of elimination rules. The implementation presents some interesting features such as performance, extensibility and ease of use, confirmed by an initial empirical assessment. In fact, this method has been more useful to us than a theorem prover in some real-world case studies.

To date, the main feature missed in the method is a seamless integration with an existing theorem prover. It would help users to prove that elimination theorems are valid, to prove properties of specifications prior to load them on Fastest, and to keep a minimal elimination theorem library.

Besides, we want to analyze whether and when running the pruning algorithm in distributed mode is faster than in application mode.

References

[1] P. Stocks and D. Carrington, "A Framework for Specification-Based Testing," *IEEE Transactions on Software Engineering*, vol. 22, no. 11, pp. 777–793, Nov. 1996.

[2] M. Cristiá and P. Rodríguez Monetti, "Implementing and applying the Stocks-Carrington framework for model-based testing," in *ICFEM*, ser. Lecture Notes in Computer Science, K. Breitman and A. Cavalcanti, Eds., vol. 5885. Springer, 2009, pp. 167–185.

[3] P. Stocks, "Applying formal methods to software testing," Ph.D. dissertation, Department of Computer Science, University of Queensland, 1993.

[4] I. Maccoll and D. Carrington, "Extending the Test Template Framework," in *Proceedings of the Third Northern Formal Methods Workshop*, 1998.

[5] M. Saaltink, "The Z/EVES mathematical toolkit version 2.2 for Z/EVES version 1.5," ORA Canada, Tech. Rep., 1997.

[6] J. M. Spivey, *The Z Notation: A Reference Manual*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1989.

[7] M. Saaltink, "The Z/EVES System," in *ZUM '97: The Z Formal Specification Notation*, J. Bowen, M. Hinchey, and D. Till, Eds., 1997, pp. 72–85.

[8] P. Malik and M. Utting, "CZT: A Framework for Z Tools," in *ZB. Lecture*. Springer, 2005, pp. 65–84.

[9] ISO, "Information Technology – Z Formal Specification Notation – Syntax, Type System and Semantics," International Organization for Standardization, Tech. Rep. ISO/IEC 13568, 2002. [Online]. Available: http://standards.iso.org/ittf/PubliclyAvailableStandards/c021573_ISO_IEC_13568_2002(E).zip

[10] M. Cristiá, P. Rodríguez Monetti, and P. Albertengo, "The Fastest 1.3.5 User's Guide," Flowgate Consulting, Tech. Rep., 2010. [Online]. Available: http://www.flowgate.net

[11] Sun Corp., "Class Pattern," http://java.sun.com/javase/6/docs/api/java/util/regex/Pattern.html.

[12] S. Helke, T. Neustupny, and T. Santen, "Automating Test Case Generation from Z Specifications with Isabelle," in *Lecture Notes in Computer Science*. Springer-Verlag, 1997, pp. 52–71.

[13] Kolyang, T. Santen, and B. Wolff, "A structure preserving encoding of Z in Isabelle/HOL," in *TPHOLs '96: Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics*. London, UK: Springer-Verlag, 1996, pp. 283–298.

[14] R. Büssow and M. Weber, "A steam-boiler control specification with Statecharts and Z," in *Formal Methods for Industrial Applications, Specifying and Programming the Steam Boiler Control*. London, UK: Springer-Verlag, 1996, pp. 109–128.

[15] J. Dick and A. Faivre, "Automating the Generation and Sequencing of Test Cases from Model-Based Specifications," in *FME '93: Proceedings of the First International Symposium of Formal Methods Europe on Industrial-Strength Formal Methods*. London, UK: Springer-Verlag, 1993, pp. 268–284.

[16] R. M. Hierons and et.al., "Using formal specifications to support testing," *ACM Comput. Surv.*, vol. 41, no. 2, pp. 1–76, 2009.

[17] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes, "Generating finite state machines from abstract state machines," in *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*. New York, NY, USA: ACM, 2002, pp. 112–122.

[18] C. Meudec, "Automatic generation of software tests from formal specifications," Ph.D. dissertation, Queen's University of Belfast, Northern Ireland, UK, 1997.

[19] R.-K. Doong and P. G. Frankl, "Case studies on testing object-oriented programs," in *TAV4: Proceedings of the symposium on Testing, analysis, and verification*. New York, NY, USA: ACM, 1991, pp. 165–177.

[20] ——, "The ASTOOT approach to testing object-oriented programs," *ACM Trans. Softw. Eng. Methodol.*, vol. 3, no. 2, pp. 101–130, 1994.