FASTEST
Automating Software Testing

Maximiliano Cristiá
mcristia@flowgate.net

Pablo Rodríguez Monetti
prodriguez@flowgate.net

Flowgate Security Consulting
Rosario – Argentina

February, 2009

1 Introduction to Model-Based Testing

- Testing is the most costly phase of a software development project.
- We can use formal methods to make testing almost automatic.
- Model-based testing uses a formal specification to generate test cases and to verify whether they found errors or not.
- The following picture depicts the basic model-based testing methodology, which is based on:
  - P. Stocks, “Applying formal methods to software testing,” Ph.D. dissertation, Department of Computer Science, University of Queensland, 1993.
  - H. M. Hrcher and J. Peleska, “Using formal specifications to support software testing,” *Software Quality Journal*, vol. 4, pp. 309–327, 1995.
  - 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.

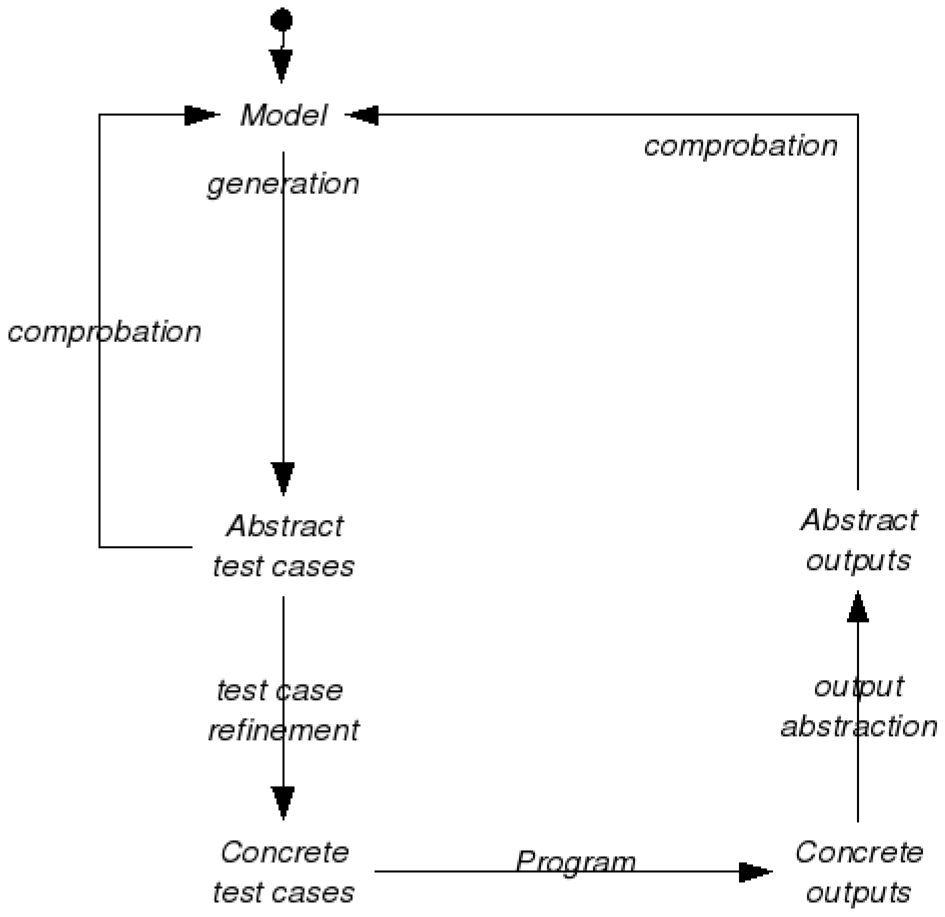

Figure 1: Model-based testing methodology: general view.

- The following figure shows with more detail how the *generation* phase is composed.
- The current version of FASTEST implements the *generation* phase but it does not include *pruning*.

  In a couple of month we’ll release a version implementing also the *test case refinement* phase.

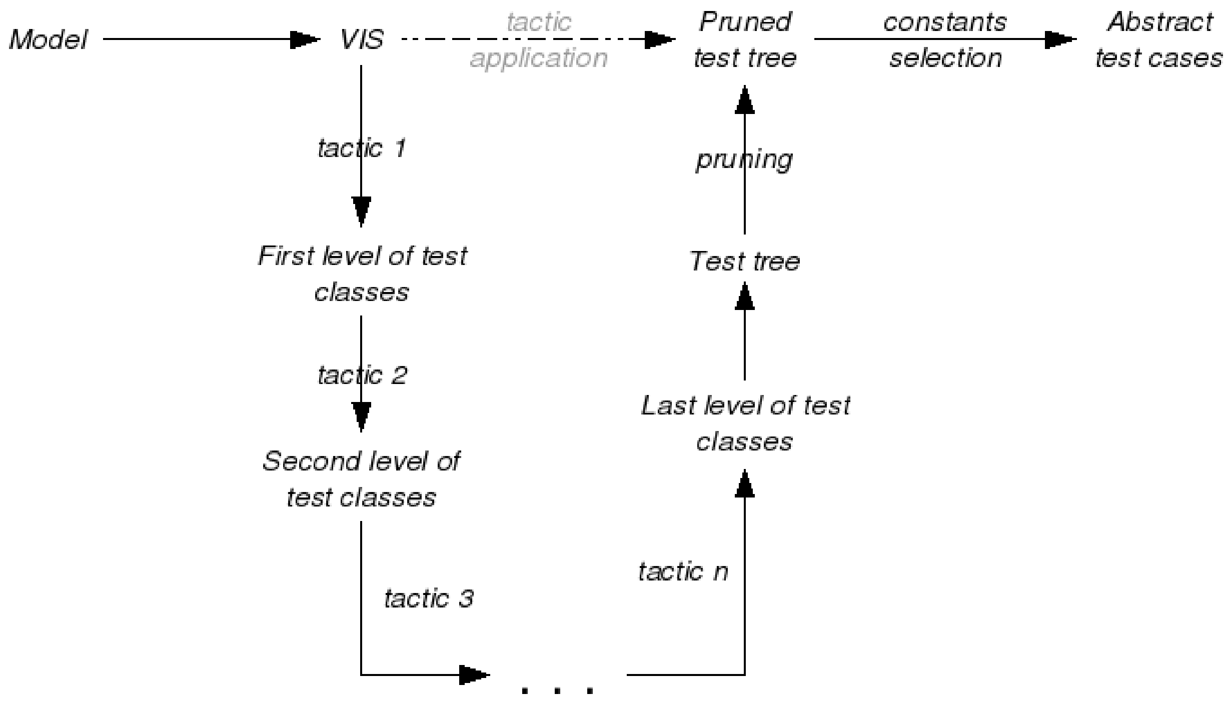

Figure 2: Model-based testing methodology: detailed view of the *generation* phase and the *tactic application* step.

- Testing tactics[1] are different ways of partitioning a set of abstract test cases.
  - Disjunctive Normal Form, Standard Partitions, Sub-domain Propagation, Cause-Effect, Specification Mutation, etc.
- Each tactic is applied over the last test tree level.
- Test classes are sets of abstract test cases defined by comprehension, then each test class is identified by a predicate.
- Test classes' predicates obey the relationship depicted in Figure 3.

- Abstract test cases are selected only from the leafs.
- As a consequence, the deeper the tree, the more accurate and discovering the test cases.
- Pruning the initial test tree saves time because many leafs are in fact empty sets (they represent impossible situations).
- A typical test tree with four levels of test classes is depicted in Figure 4.

[1]Stocks and Carrington use strategies instead of tactics.

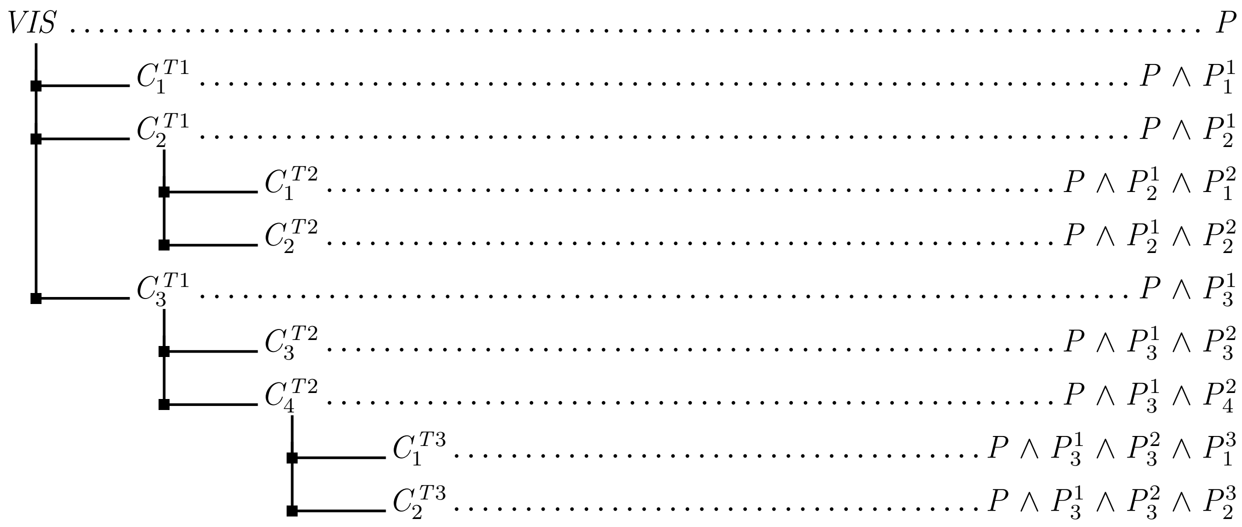
VIS ... P
C_1^{T1} ... $P \wedge P_1^1$
C_2^{T1} ... $P \wedge P_2^1$
C_1^{T2} ... $P \wedge P_2^1 \wedge P_1^2$
C_2^{T2} ... $P \wedge P_2^1 \wedge P_2^2$
C_3^{T1} ... $P \wedge P_3^1$
C_3^{T2} ... $P \wedge P_3^1 \wedge P_3^2$
C_4^{T2} ... $P \wedge P_3^1 \wedge P_4^2$
C_1^{T3} ... $P \wedge P_3^1 \wedge P_3^2 \wedge P_1^3$
C_2^{T3} ... $P \wedge P_3^1 \wedge P_3^2 \wedge P_2^3$

Figure 3: The predicate of a test class at some level is the conjunction of the predicate of its parent test class and a its own predicate.

2 FASTEST Architecture in a Nutshell

- FASTEST architecture was guided by:
  - Performance, because calculating thousands of test cases can be very time consuming;
  - Modificability, because we don't know yet what features industry could need; and
  - Documentability, because to test must mean to document.
- Hence, we combined two different architectural styles and we used open formats and tools in our interfaces:
  - Client/Server, so we distribute test case calculation;
  - Implicit Invocation, so we can add, modify and remove components as new and more sophisticated requirements arise;
  - Latex is used to read specifications and to generate test trees, test classes, abstract test cases, etc.;
  - FASTEST is integrated with the Community Z Tools (CZT, `http://czt.sourceforge.net/`), to avoid duplicate efforts in programming core modules.
- FASTEST' process structure is shown in Figure 5.
  - Clients interact with the user, generate test trees and run test cases.
  - All of the other functions are performed on the servers.
  - The knowledge base server stores testing tactics, abstract test cases, refinement functions, etc. so testers can use them for re-testing within a given project and for testing in different projects.

    Currently this server is no fully implemented.

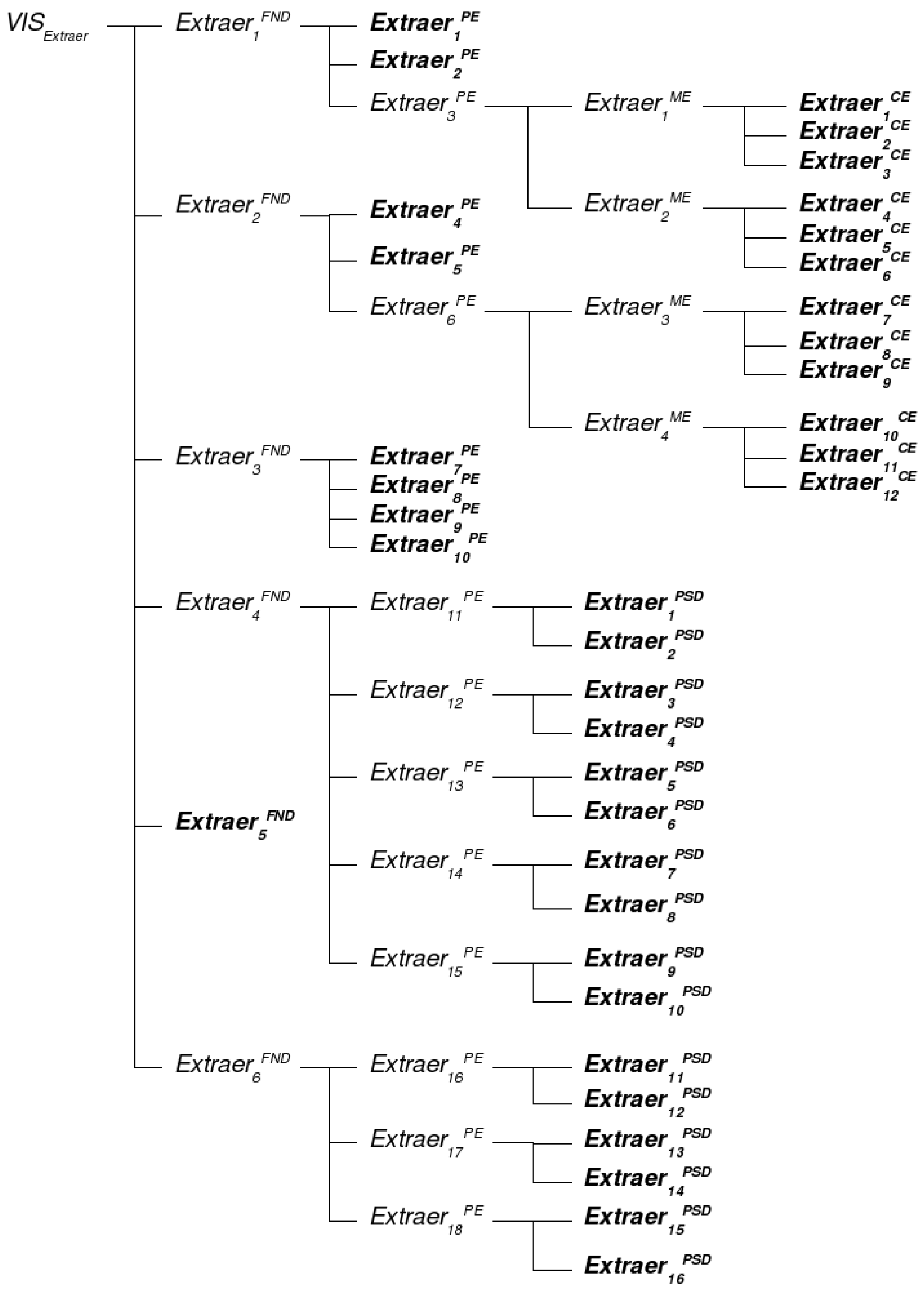

Figure 4: Example of a test tree with four levels of test classes.

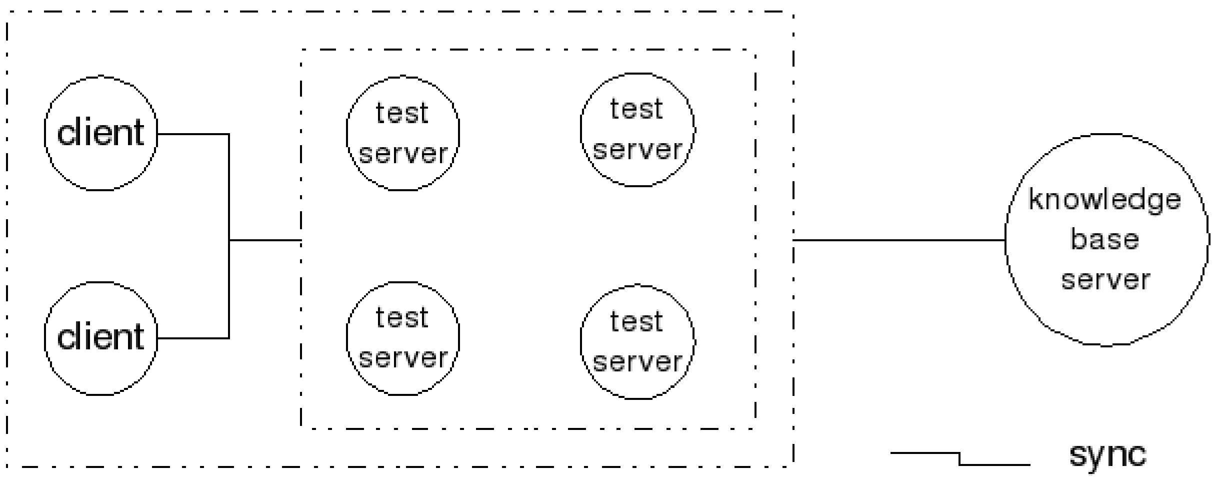

Figure 5: FASTEST's process structure is composed by a number of clients and servers, and just one instance of a knowledge-base server.

3 An Example

- We have said that model-based testing takes as input a formal model of the system to be tested.
- FASTEST uses Z specifications.

  The current version does not support the full language. For instance, schema composition, schema piping, the theta operator, schema as types, axiomatic definitions and so on, are not supported yet.
- The first goal is to apply FASTEST to unit testing; then we'll try to extend it to integration and system testing.
- Then, say we have to test a function that must keep the highest readings from a set of sensors.

3.1 The Z Formal Specification Language

- Z is a formal notation useful to specify systems that will use complex data structures and will apply complex transformations over them.
- It's a textual language based on first order logic and discrete mathematics.
- A Z model is a state machine where states are defined by state variables and transitions are predicates over those variables.
- The Z formal specification for the function mentioned above is as follows.
- Note that we do not include state invariants in the state schema, but instead we find that writing them as proof obligations (like in the B method or TLA+) is a better practice.

  Actually we did not write the invariant in this example in order to simplify the presentation.

  $[SENSOR]$

  $MaxReadings == [smax : SENSOR \rightarrow\!\!\!\!\!\!+ \mathbb{Z}]$

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\_\, \mathit{KeepMaxReadingOk} \,\_\_\_\_\_\_\_\_} \\
\Delta \mathit{MaxReadings} \\
s? : \mathit{SENSOR};\ r? : \mathbb{Z} \\
\hline
s? \in \operatorname{dom} \mathit{smax} \\
\mathit{smax}\ s? < r? \\
\mathit{smax}' = \mathit{smax} \oplus \{s? \mapsto r?\} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\_\, \mathit{KeepMaxReadingE2} \,\_\_\_\_\_\_\_\_} \\
\Xi \mathit{MaxReadings} \\
s? : \mathit{SENSOR};\ r? : \mathbb{Z} \\
\hline
s? \in \operatorname{dom} \mathit{smax} \\
r? \leq \mathit{smax}\ s? \\
\hline
\end{array}$$

$$\begin{array}{l}
\mathit{KeepMaxReadingE1} == \\
\qquad [\Xi \mathit{MaxReadings};\ s? : \mathit{SENSOR} \mid \\
\qquad s? \notin \operatorname{dom} \mathit{smax}]
\end{array}$$

$$\begin{array}{l}
\mathit{KeepMaxReading} == \\
\qquad \mathit{KeepMaxReadingOk} \\
\qquad \lor \mathit{KeepMaxReadingE1} \\
\qquad \lor \mathit{KeepMaxReadingE2}
\end{array}$$

3.2 Generating the Test Tree

- The first step is to use FASTEST to generate the test tree.
- The test tree depends on the tactics you apply and the order they are applied.
- In this case we apply two testing tactics:
  - Disjunctive Normal Form (DNF); it's applied by default.
  - Standard Partitions (SP), applied to the expression $\mathit{smax}\ s? < r?$ present in schema *KeepMaxReadingOk*.
- Then, the test tree is the one shown in Figure 6.
- Each node in the test tree is also a Z schema, each of which describes the conditions for selecting test cases.

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\_\, \mathit{KeepMaxReading\_VIS} \,\_\_\_\_\_\_\_\_} \\
\mathit{smax} : \mathit{SENSOR} \pfun \mathbb{Z} \\
s? : \mathit{SENSOR} \\
r? : \mathbb{Z} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\_\, \mathit{KeepMaxReading\_SP\_1} \,\_\_\_\_\_\_\_\_} \\
\mathit{smax} : \mathit{SENSOR} \pfun \mathbb{Z} \\
s? : \mathit{SENSOR} \\
r? : \mathbb{Z} \\
\hline
s? \in \operatorname{dom} \mathit{smax} \\
\mathit{smax}\ s? < r? \\
\mathit{smax}\ s? < 0 \\
r? < 0 \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\_\, \mathit{KeepMaxReading\_DNF\_1} \,\_\_\_\_\_\_\_\_} \\
\mathit{smax} : \mathit{SENSOR} \pfun \mathbb{Z} \\
s? : \mathit{SENSOR} \\
r? : \mathbb{Z} \\
\hline
s? \in \operatorname{dom} \mathit{smax} \\
\mathit{smax}\ s? < r? \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\_\, \mathit{KeepMaxReading\_SP\_2} \,\_\_\_\_\_\_\_\_} \\
\mathit{smax} : \mathit{SENSOR} \pfun \mathbb{Z} \\
s? : \mathit{SENSOR} \\
r? : \mathbb{Z} \\
\hline
s? \in \operatorname{dom} \mathit{smax} \\
\mathit{smax}\ s? < r? \\
\mathit{smax}\ s? < 0 \\
r? = 0 \\
\hline
\end{array}$$

```
KeepMaxReading_VIS
├── KeepMaxReading_DNF_1
│   ├── KeepMaxReading_SP_1
│   ├── KeepMaxReading_SP_2
│   ├── KeepMaxReading_SP_3
│   ├── KeepMaxReading_SP_4
│   ├── KeepMaxReading_SP_5
│   ├── KeepMaxReading_SP_6
│   ├── KeepMaxReading_SP_7
│   ├── KeepMaxReading_SP_8
│   └── KeepMaxReading_SP_9
├── KeepMaxReading_DNF_2
│   ├── KeepMaxReading_SP_10
│   ├── KeepMaxReading_SP_11
│   ├── KeepMaxReading_SP_12
│   ├── KeepMaxReading_SP_13
│   ├── KeepMaxReading_SP_14
│   ├── KeepMaxReading_SP_15
│   ├── KeepMaxReading_SP_16
│   ├── KeepMaxReading_SP_17
│   └── KeepMaxReading_SP_18
└── KeepMaxReading_DNF_3
    ├── KeepMaxReading_SP_19
    ├── KeepMaxReading_SP_20
    ├── KeepMaxReading_SP_21
    ├── KeepMaxReading_SP_22
    ├── KeepMaxReading_SP_23
    ├── KeepMaxReading_SP_24
    ├── KeepMaxReading_SP_25
    ├── KeepMaxReading_SP_26
    └── KeepMaxReading_SP_27
```

Figure 6: Testing tree for *KeepMaxReading*.

KeepMaxReading_SP_3

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

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

KeepMaxReading_SP_4

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

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

KeepMaxReading_SP_5

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

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

KeepMaxReading_SP_6

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

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

KeepMaxReading_SP_7

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

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

KeepMaxReading_SP_8

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

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

KeepMaxReading_SP_9

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

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

KeepMaxReading_DNF_2

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

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

KeepMaxReading_SP_10

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

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

KeepMaxReading_SP_11

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

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

$$\begin{array}{|l}\multicolumn{1}{l}{KeepMaxReading\_SP\_12}\\ \hline smax : SENSOR \pfun \mathbb{Z}\\ s? : SENSOR\\ r? : \mathbb{Z}\\ \hline s? \notin \operatorname{dom} smax\\ smax~s? < 0\\ r? > 0\\ \hline\end{array}$$

$$\begin{array}{|l}\multicolumn{1}{l}{KeepMaxReading\_SP\_13}\\ \hline smax : SENSOR \pfun \mathbb{Z}\\ s? : SENSOR\\ r? : \mathbb{Z}\\ \hline s? \notin \operatorname{dom} smax\\ smax~s? = 0\\ r? < 0\\ \hline\end{array}$$

$$\begin{array}{|l}\multicolumn{1}{l}{KeepMaxReading\_SP\_14}\\ \hline smax : SENSOR \pfun \mathbb{Z}\\ s? : SENSOR\\ r? : \mathbb{Z}\\ \hline s? \notin \operatorname{dom} smax\\ smax~s? = 0\\ r? = 0\\ \hline\end{array}$$

$$\begin{array}{|l}\multicolumn{1}{l}{KeepMaxReading\_SP\_15}\\ \hline smax : SENSOR \pfun \mathbb{Z}\\ s? : SENSOR\\ r? : \mathbb{Z}\\ \hline s? \notin \operatorname{dom} smax\\ smax~s? = 0\\ r? > 0\\ \hline\end{array}$$

$$\begin{array}{|l}\multicolumn{1}{l}{KeepMaxReading\_SP\_16}\\ \hline smax : SENSOR \pfun \mathbb{Z}\\ s? : SENSOR\\ r? : \mathbb{Z}\\ \hline s? \notin \operatorname{dom} smax\\ smax~s? > 0\\ r? < 0\\ \hline\end{array}$$

$$\begin{array}{|l}\multicolumn{1}{l}{KeepMaxReading\_SP\_17}\\ \hline smax : SENSOR \pfun \mathbb{Z}\\ s? : SENSOR\\ r? : \mathbb{Z}\\ \hline s? \notin \operatorname{dom} smax\\ smax~s? > 0\\ r? = 0\\ \hline\end{array}$$

$$\begin{array}{|l}\multicolumn{1}{l}{KeepMaxReading\_SP\_18}\\ \hline smax : SENSOR \pfun \mathbb{Z}\\ s? : SENSOR\\ r? : \mathbb{Z}\\ \hline s? \notin \operatorname{dom} smax\\ smax~s? > 0\\ r? > 0\\ \hline\end{array}$$

$$\begin{array}{|l}\multicolumn{1}{l}{KeepMaxReading\_DNF\_3}\\ \hline smax : SENSOR \pfun \mathbb{Z}\\ s? : SENSOR\\ r? : \mathbb{Z}\\ \hline s? \in \operatorname{dom} smax\\ r? \leq smax~s?\\ \hline\end{array}$$

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

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

$$\begin{array}{|l}
\multicolumn{1}{l}{KeepMaxReading\_SP\_21} \\ \hline
smax : SENSOR \pfun \mathbb{Z} \\
s? : SENSOR \\
r? : \mathbb{Z} \\ \hline
s? \in \operatorname{dom} smax \\
r? \leq smax~s? \\
smax~s? < 0 \\
r? > 0 \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{KeepMaxReading\_SP\_22} \\ \hline
smax : SENSOR \pfun \mathbb{Z} \\
s? : SENSOR \\
r? : \mathbb{Z} \\ \hline
s? \in \operatorname{dom} smax \\
r? \leq smax~s? \\
smax~s? = 0 \\
r? < 0 \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{KeepMaxReading\_SP\_23} \\ \hline
smax : SENSOR \pfun \mathbb{Z} \\
s? : SENSOR \\
r? : \mathbb{Z} \\ \hline
s? \in \operatorname{dom} smax \\
r? \leq smax~s? \\
smax~s? = 0 \\
r? = 0 \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{KeepMaxReading\_SP\_24} \\ \hline
smax : SENSOR \pfun \mathbb{Z} \\
s? : SENSOR \\
r? : \mathbb{Z} \\ \hline
s? \in \operatorname{dom} smax \\
r? \leq smax~s? \\
smax~s? = 0 \\
r? > 0 \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{KeepMaxReading\_SP\_25} \\ \hline
smax : SENSOR \pfun \mathbb{Z} \\
s? : SENSOR \\
r? : \mathbb{Z} \\ \hline
s? \in \operatorname{dom} smax \\
r? \leq smax~s? \\
smax~s? > 0 \\
r? < 0 \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{KeepMaxReading\_SP\_26} \\ \hline
smax : SENSOR \pfun \mathbb{Z} \\
s? : SENSOR \\
r? : \mathbb{Z} \\ \hline
s? \in \operatorname{dom} smax \\
r? \leq smax~s? \\
smax~s? > 0 \\
r? = 0 \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{KeepMaxReading\_SP\_27} \\ \hline
smax : SENSOR \pfun \mathbb{Z} \\
s? : SENSOR \\
r? : \mathbb{Z} \\ \hline
s? \in \operatorname{dom} smax \\
r? \leq smax~s? \\
smax~s? > 0 \\
r? > 0 \\ \hline
\end{array}$$

3.3 Finding Abstract Test Cases

- Now, FASTEST tries to find one abstract test case in each leaf of the testing tree.
- Here FASTEST generates a finite model over which it evaluates each leaf's predicate.
  - If some element of the finite model satisfies a test class' predicate, then we have found an abstract test case in that class.
  - If no element in the finite model satisfies the predicate, it can be due to two causes:
    - The predicate is a contradiction (during pruning FASTEST could not determine that).
    - The finite model is not a appropriate.

Leafs	**Found**	**Failed**		
		Possible	**Impossible**	**Undefined**
27	**11**	**0**	**7**	**9**

Table 1: Summary

- Table 1 gives a summary of the search for abstract test cases for the present example.
  - **(Possible)** In a first attempt, FASTEST failed to find abstract test cases for

    KeepMaxReading_SP_1

    $$\begin{array}{l} smax : SENSOR \pfun \mathbb{Z} \\ s? : SENSOR \\ r? : \mathbb{Z} \\ \hline s? \in \operatorname{dom} smax \\ smax\ s? < r? \\ smax\ s? < 0 \\ r? < 0 \end{array}$$

    KeepMaxReading_SP_9

    $$\begin{array}{l} smax : SENSOR \pfun \mathbb{Z} \\ s? : SENSOR \\ r? : \mathbb{Z} \\ \hline s? \in \operatorname{dom} smax \\ smax\ s? < r? \\ smax\ s? > 0 \\ r? > 0 \end{array}$$

    because the default finite set for $\mathbb{Z}$ is $\{-1, 0, 1\}$.

    However, if before instructing FASTEST to find abstract test cases, we instruct it to build other finite models for these test classes, then it will find the corresponding abstract test cases.

    On the other hand, you have to take into consideration that FASTEST needs almost 9 hours to find a test case for *KeepMaxReading_SP_9*. Future versions of FASTEST will dramatically improve on this.
  - **(Impossible)** Most, if not all, the empty test classes can automatically be eliminated when the pruning step will be implemented.
  - **(Undefined)** The nine undefined cases correspond to test classes where $s? \notin \operatorname{dom} smax$ is part of the predicate, because the predicate contains also the expression $smax\ s?$ which is undefined when $s? \notin \operatorname{dom} smax$.

    For these cases we have two options (we're still investigating which is the best). If P is a predicate depending on $f\ x$ and $x \notin \operatorname{dom} f$, then we can consider either:
    * That $P(f\ x)$ is false, in which case the test class containing P is empty and should be pruned from the test tree.
    * That $P(f\ x)$ is true regardless the value of $f\ x$, in which case FASTEST should try to find an abstract test case in this test class but considering only the other predicates that define the class.

    In any case it's rather easy to improve on these cases making FASTEST find either one or nine more abstract test cases.
  - **In summary, future versions of FASTEST will automatically yield between 12 to 20 test cases for an operation like *KeepMaxReading*.**
- The extended test tree, i.e. the test tree containing also the abstract test cases of the test classes for which it was possible to find one, is shown in Figure 7.
- Again, each abstract test case is a Z schema, but now each state and input variable equals a constant value.

```
KeepMaxReading_VIS
├── KeepMaxReading_DNF_1
│   ├── KeepMaxReading_SP_1
│   │   └── KeepMaxReading_SP_1_TCASE
│   ├── KeepMaxReading_SP_2
│   │   └── KeepMaxReading_SP_2_TCASE
│   ├── KeepMaxReading_SP_3
│   │   └── KeepMaxReading_SP_3_TCASE
│   ├── KeepMaxReading_SP_4
│   ├── KeepMaxReading_SP_5
│   ├── KeepMaxReading_SP_6
│   │   └── KeepMaxReading_SP_6_TCASE
│   ├── KeepMaxReading_SP_7
│   ├── KeepMaxReading_SP_8
│   └── KeepMaxReading_SP_9
│       └── KeepMaxReading_SP_9_TCASE
├── KeepMaxReading_DNF_2
│   ├── KeepMaxReading_SP_10
│   ├── KeepMaxReading_SP_11
│   ├── KeepMaxReading_SP_12
│   ├── KeepMaxReading_SP_13
│   ├── KeepMaxReading_SP_14
│   ├── KeepMaxReading_SP_15
│   ├── KeepMaxReading_SP_16
│   ├── KeepMaxReading_SP_17
│   └── KeepMaxReading_SP_18
└── KeepMaxReading_DNF_3
    ├── KeepMaxReading_SP_19
    │   └── KeepMaxReading_SP_19_TCASE
    ├── KeepMaxReading_SP_20
    ├── KeepMaxReading_SP_21
    ├── KeepMaxReading_SP_22
    │   └── KeepMaxReading_SP_22_TCASE
    ├── KeepMaxReading_SP_23
    │   └── KeepMaxReading_SP_23_TCASE
    ├── KeepMaxReading_SP_24
    ├── KeepMaxReading_SP_25
    │   └── KeepMaxReading_SP_25_TCASE
    ├── KeepMaxReading_SP_26
    │   └── KeepMaxReading_SP_26_TCASE
    └── KeepMaxReading_SP_27
        └── KeepMaxReading_SP_27_TCASE
```

Figure 7: The extended test tree includes also the schema boxes representing test cases hanging from those leafs for which FASTES was able to find a test case.

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

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

$$\begin{array}{|l}\multicolumn{1}{l}{KeepMaxReading\_SP\_3\_TCASE}\\ KeepMaxReading\_SP\_3\\ \hline r? = 1\\ smax = \{(sensor0, -1)\}\\ s? = sensor0\\ \hline\end{array}$$

$$\begin{array}{|l}\multicolumn{1}{l}{KeepMaxReading\_SP\_6\_TCASE}\\ KeepMaxReading\_SP\_6\\ \hline r? = 1\\ smax = \{(sensor0, 0)\}\\ s? = sensor0\\ \hline\end{array}$$

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

$$\begin{array}{|l}\multicolumn{1}{l}{KeepMaxReading\_SP\_19\_TCASE}\\ KeepMaxReading\_SP\_19\\ \hline r? = -1\\ smax = \{(sensor0, -1)\}\\ s? = sensor0\\ \hline\end{array}$$

$$\begin{array}{|l}\multicolumn{1}{l}{KeepMaxReading\_SP\_22\_TCASE}\\ KeepMaxReading\_SP\_22\\ \hline r? = -1\\ smax = \{(sensor0, 0)\}\\ s? = sensor0\\ \hline\end{array}$$

$$\begin{array}{|l}\multicolumn{1}{l}{KeepMaxReading\_SP\_23\_TCASE}\\ KeepMaxReading\_SP\_23\\ \hline r? = 0\\ smax = \{(sensor0, 0)\}\\ s? = sensor0\\ \hline\end{array}$$

$$\begin{array}{|l}\multicolumn{1}{l}{KeepMaxReading\_SP\_25\_TCASE}\\ KeepMaxReading\_SP\_25\\ \hline r? = -1\\ smax = \{(sensor0, 1)\}\\ s? = sensor0\\ \hline\end{array}$$

$$\begin{array}{|l}\multicolumn{1}{l}{KeepMaxReading\_SP\_26\_TCASE}\\ KeepMaxReading\_SP\_26\\ \hline r? = 0\\ smax = \{(sensor0, 1)\}\\ s? = sensor0\\ \hline\end{array}$$

$$\begin{array}{|l}\multicolumn{1}{l}{KeepMaxReading\_SP\_27\_TCASE}\\ KeepMaxReading\_SP\_27\\ \hline r? = 1\\ smax = \{(sensor0, 1)\}\\ s? = sensor0\\ \hline\end{array}$$

3.4 The FASTEST Script

The FASTEST script to do all the previous work is (assuming that file `sensors-simp.tex` is in the same directory than `fastest.jar`, i.e. the installation directory):

```
$> java -jar fastest.jar
FASTest version 1.0, (C) 2008, Flowgate Security Consulting
FASTest>loadspec sensors-simp.tex
FASTest>selop KeepMaxReading
FASTest>addtactic KeepMaxReading SP < smax~s? < r?
FASTest>settcasestrategy Iterative KeepMaxReading_SP_1 4 -int Zero
FASTest>settcasestrategy Iterative KeepMaxReading_SP_9 5 -int Zero
FASTest>genalltca
FASTest>showtt
FASTest>showsch -tca
```

The commands used in this script as well as other commands are explained in the next section.

3.4.1 Adding One More Level to the Test Tree

By applying another standard partition (to $\oplus$ over $smax \oplus \{s? \mapsto r?\}$) we get a 243 leafs test tree. The command to apply the standard partition is as follows.

```
FASTest>addtactic KeepMaxReading SP \oplus smax \oplus \{ s? \mapsto r? \}
FASTest>genalltt
```

4 User's Manual

4.1 Installing and executing FASTEST

FASTEST works in both Linux and MS-Windows environments. It requires a Java SE Runtime Environment 1.6 or superior. To install the tool, just uncompress and unarchive the file fastest.tar.gz in any folder of your choice. To run FASTEST, open a command window, move to the folder where you installed it, and run the following command:

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

FASTEST then opens a FTP-like user interface where you can run commands. To leave the program you can issue `quit`; also note that `Ctrl+C` kills the program making all your data and commands to be lost –future versions will be more robust.

4.2 Steps of a testing campaign

Roughly speaking, currently, a FASTEST testing campaign can be decomposed in the following steps:

1. Load the specification
2. Select the operations to be tested
3. [Optional] Select a list of testing tactics to be applied to each operation
4. [Optional] Instruct FASTEST on how to generate finite models to find abstract test cases
5. Run the command to either calculate all the test trees or all the abstract test cases

Also, at any time you can run commands to explore the specification and the test trees.

Although the third step is optional, in only very trivial situations you will not use it. In fact, it is perhaps the most relevant step of all since it will determine how revealing and leafy your test trees are going to be.

Note that, currently, you must run all the commands of steps 3 and 4 before step 5 because once you have launched the commands of the last step, FASTEST will not allow you to run more commands of the previous steps.

4.3 Command description

4.3.1 Basic commands

- The command `help` might provide some help about the other available commands.
- The specification is loaded by executing `loadspec` followed by a file name; you must write the full path to the file if it is not located in the installation directory.

  It is assumed that the file is a legal Latex file.

  If the specification contains syntactic or type errors it will not be loaded and the errors will be informed.
- `selop` selects an operation (of the Z model) to be "tested". You can select as many operation as you wish, but you have to run the command for each of them.

4.3.2 Selecting and defining testing tactics

The command `addtactic` allows you to add a testing tactic to the list of tactics to be applied to a particular (previously selected) operation. Initially, the list of tactics of any operation includes only the tactic named Disjunctive Normal Form (see below).

The command syntax is rather complex because it depends on the tactic you want to apply. The base syntax is

```
addtactic op_name tactic_name parameters
```

where `op_name` is the name of a selected (Z) operation, `tactic_name` is the name of a tactic supported by FASTEST, and `parameters` is a list of parameters that depends on the tactic.

Currently the following tactics are supported: Disjunctive Normal Form (but this is applied by default and you must not add it), `SP` for Standard Partition and `FT` for Free Type. Their syntax and semantics are as follows.

- Disjunctive Normal Form. This tactic is applied by default and you must not select it with `addtactic`. By applying this tactic the operation is written in Disjunctive Normal Form and the *VIS* is divided in as many test classes as terms are in the operation's predicate. The predicate added to each class is the precondition of one of the terms in the operation's predicate.
- Standard Partition. This tactic uses a predefined partition of some mathematical operator (see "Standard domains for Z operators" at page 165 of Stocks' PhD thesis).

  Take a look at file `./fastest/lib/conf/stdpartition.spf` to see what standard partitions are delivered with FASTEST and how to define new ones (you need to restart FASTEST if you do so).

  In this case the command is

  ```
  addtactic op_name SP operator expression
  ```

  where `operator` is the Latex-CZT string of a Z operator and `expression` is a Z expression written in Latex-CZT. It is assumed that `operator` appears in the `expression` and this in turn appears in the predicate of the selected operation.

  Hence, you can apply this tactic to different operators and different expressions of the same operation.

- Free Type. This tactic generates as many test classes as elements a free type (enumerated) has. In other words if your model has type $COLOUR ::= red \mid blue \mid green$ and some operation uses c of type $COLOUR$, then by applying this tactic you will get three test classes: one in which c equals red, the other in which c equals $blue$, and the third where c equals $green$.

  In this case the command is

  ```
  addtactic op_name FT variable
  ```

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

  Currently, Free Type works only if the free type is actually an *enumerated* type, i.e. a type without induction.

4.3.3 Defining finite models

As we explained in section 3.3, FASTEST calculates abstract test cases by generating a finite model for each leaf test class. This finite model is the Cartesian product between one finite set for each variable appearing in the predicate of the test class. Clearly, the bigger the finite model the more chances to find an abstract test case, but the more time it will take. Then, the best strategy here is to help FASTEST to select the most promising and smallest finite model.

FASTEST allows the user to define the strategy for generating a finite model at the test class level; in other words, you can define a different strategy for each test class.

If the user do not take any explicit action, FASTEST generates a default finite model that we believe is the best option in most situations. These finite models are constructed from specific finite sets selected for all basic and enumerated types, $\mathbb{N}$ and $\mathbb{Z}$. All these finite sets have three elements, except for enumerated types for which FASTEST defines sets with the same number of elements than the type. The default sets for $\mathbb{N}$ and $\mathbb{Z}$ are $\{0, 1, 2\}$ and $\{-1, 0, 1\}$, respectively.

Currently, FASTEST allows the user to select the number of elements of the finite sets of type $\mathbb{N}$, $\mathbb{Z}$ and the given types, that will be used to generate the finite model. Also, for $\mathbb{Z}$ and $\mathbb{N}$ it is possible to select what elements the finite sets are going to include. The basic command for selecting the strategy is as follows:

```
settcasestrategy strategy_name tclass_name set_size parameters
```

where `strategy_name` can be either `Complete` or `Iterative`, but we strongly recommend to always use `Iterative`. `tclass_name` is the name of a test class in some test tree and `set_size` is a natural number telling how many elements the finite sets of type $\mathbb{N}$ or $\mathbb{Z}$, present in that class, will have. It is very important to remark that `set_size` has a tremendous impact on the size of the finite model for the test class because functions, relations, sequences and other structured types are built from those sets. For instance, if a test class contains a variable of type $\mathbb{Z} \rightarrow\!\!\!\!\!\!+ \mathbb{Z}$ and the user has set `set_size` to, say, 5, then FASTEST will generate all the partial functions from {-2,-1,0,1,2} onto itself, and then it will evaluate the test class predicate over all of the combinations between the values of all the variables in the VIS. This can be a really huge number of combinations. It can take days to perform all of these evaluations if the predicate is moderately complex.

The last parameter, namely `parameters`, can be used only when `strategy_name` is equal to `Iterative`. In this case, the command is:

```
settcasestrategy Iterative tclass_name set_size -int OPT | -nat OPT
```

where `OPT` can be one of `Given`, `Zero` or `Seeds`. The `int` (`nat`) option specifies what elements the finite sets of type $\mathbb{Z}$ ($\mathbb{N}$) will have. As you can see there are three different choices of how those sets will be generated, with slightly different meanings depending on whether they are applied to `int` or `nat`. In the following paragraphs consider that the user has set `set_size` to n, with $n > 0$, and that the rules for sets of type $\mathbb{Z}$ ($\mathbb{N}$) apply only if `int` (`nat`) was used.

`Zero` A set of type $\mathbb{Z}$ in the test class will be reduced to the following interval:

$$\begin{cases} 0 & \text{if} \quad n = 1 \\ -(n \operatorname{div} 2) \mathrel{..} n \operatorname{div} 2 & \text{if} \quad n > 1 \land n \bmod 2 \neq 0 \\ -(n \operatorname{div} 2) \mathrel{..} n \operatorname{div} 2 - 1 & \text{if} \quad n > 1 \land n \bmod 2 = 0 \end{cases}$$

On the other hand, a set of type $\mathbb{N}$ in the test class will be the interval $0 \mathrel{..} n$.

`Given` If this option is chosen, then FASTEST will search for constants of type $\mathbb{Z}$ or $\mathbb{N}$, depending on the selected option, present in the test class. If there are no such constants, then `Zero` is applied.

If there are such constants, then a set of type $\mathbb{Z}$ will include all of them plus the integer number one unit less than the minimum of these constants and the integer number one unit more than the maximum of these constants. While a set of type $\mathbb{N}$ will have the same property changing "integer" by "natural" and noting that if zero is one of the constants then it will be the minimum. Observe that in these last cases the size set by the user in the command is irrelevant.

These are the default rules that are applied if the user does not run any `settcasestrategy` command for a test class.

`Seeds` This option is similar to `Given`. It takes the same constants, adds the same upper and lower limits but also adds the mean value between each pair of consecutive constants found in the test class.

Same considerations than in `Give` applies if there are no $\mathbb{N}$ or $\mathbb{Z}$ constants; if zero is one of the natural numbers found in the class; and regarding the size passed by the user as a parameter.

4.4 Generating test trees and abstract test cases

- `genalltca` applies the added tactics to all the selected operations, then calculates the test tree for each operation, and finally it distributes the calculation of abstract test cases between all the registered servers. For those test classes that a `settcasestrategy` command was executed, `genalltca` will use the finite model indicated by that command.

  You cannot use the tool until the whole process terminates.

  If you interrupt the process, you will have to start it all over again.

- It is possible to calculate the test tree without generating test cases by running `genalltt` instead of `genalltca`, but if then you want to generate test cases, you will have to select all the operations again and run `genalltca`.

4.5 Exploring the specification and test trees

You can display the results of the "testing" process with commands `showspec`, `showloadedops`, `showtt` and `showsch`. All share some options, check with `help`.

- The `-u` option followed by a natural number displays the result with more or less detail (basically it expands up to some level the included schema boxes).
- The `-o` option lets you redirect the output to a file.
- `showspec` just prints the whole specification.
- `showloadedops` lists the operations present in the specification.
- `showtt` displays the test tree.
- `showsch -tcl` shows all the schema corresponding to test classes.
- `showsch -tca` shows all the schema corresponding to abstract test cases.
- In this two variants the `-p` option followed by the name of an operation, only displays test classes or abstract test cases for this operation.