# A Language for Test Case Refinement in the Test Template Framework

M. Cristiá[2,3], D. Hollmann[2], P. Albertengo[1],
C. Frydman[3], and P. Rodríguez Monetti[4]

[1] Flowgate Consulting, Rosario, Argentina
[2] CIFASIS-UNR, Rosario Argentina
[3] LSIS-UPCAM, Marseille, France
[4] FCEIA-UNR, Rosario, Argentina
mcristia@flowgate.net

**Abstract.** Model-based testing (MBT) generates test cases by analysing a formal model of the system under test (SUT). In many MBT methods, these test cases are too abstract to be executed. Therefore, an executable representation of them is necessary to test the SUT. So far, the MBT community has focused on methods that automate the generation of test cases, but less has been done in making them executable. In this paper we propose a language to specify rules that can be automatically applied to produce an executable representation of test cases generated by the Test Template Framework (TTF), a MBT method for the Z notation.

## 1 The Process of Model-Based Testing

Model-based testing (MBT) is a well-known technique aimed at testing software by analysing a formal model or specification of the system under test (SUT) [1, 2]. 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.

Figure 1 depicts a possible testing process when a MBT method is applied. So far, the MBT community has focused on the "Generation" step in which testers analyse a model of the SUT and generate test cases by applying different techniques. Test cases produced by the "Generation" step are abstract in the sense that they are written in the same language of the model, making them, in most of the MBT methods, not executable. In effect, during the "Refinement" step these abstract test cases are made executable by a process that can be called *refinement*, *concretization* or *reification*. Note that this not necessarily means that the SUT has been refined from the model; it only says that test cases must be refined. In fact, Hierons and others conclude that the relation between refinement and MBT is still a challenge that would have a very tangible benefit if solved [2]. Besides, test case refinement can require an effort equal to the 25% up to 100% of the time spent on modelling [1], so it is worth to automate this step

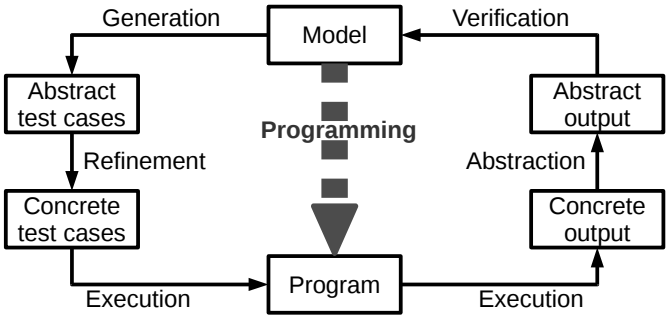


**Fig. 1.** A general description of a possible MBT process.

as much as possible. Furthermore, automating this step enables the automation of the rest of the MBT process. Once test cases have been refined they can be automatically executed by one of the many test execution environments or techniques already developed [7, 8]. Hence, the problem we try to solve is the automation of the "Refinement" step and not the automation of the "Execution" step, which has been extensively studied.

As we have said, there is a variety of MBT methods for many formal notations. Our work concentrates in the Z notation [9]. Z is a widely known formal notation based on first order logic and set theory. There are some MBT methods available for the Z notation. In [10] the authors apply category–partition; Hall [11] generates tests by analysing the test domains of Z operations; in [12] the Z information provided in a $\mu$SZ specification is used to provide sequences of transitions that covers a EFSM derived from the specification; Hierons [13] also partitions a Z operation and then derives a FSA to control how testing is performed; Horcher and Peleska [14] apply DNF to a Z operation and describe a MBT process similar to the one in Figure 1. However, we think that the Test Template Framework (TTF) [3, 15, 16] is the MBT method that takes the most of the Z notation, as we will show in Section 2. We have developed the first automatic implementation of the TTF in a tool called Fastest [17–19].

Hence, in this paper we propose a test case refinement language (TCRL) as an extension to the TTF. This TCRL does not assume that the SUT has been refined from the Z specification. In fact, if this is the case there might be better options [20]. However, our method does assumes that the SUT's source code is available. We have implemented an interpreter for this TCRL in Fastest following an architecture that allows users to automatically refine test cases to different programming languages after specifying simple refinement rules. Furthermore, the architecture makes it easy to plug in modules that implement the TCRL for programming languages not yet supported by the tool. The implementation is still a research prototype.

This article is a summary of a 65 page long reference manual describing the TCRL [21]. Therefore, due to space restrictions, here we will introduce only its most relevant features by means of some running examples. This document along

with examples to be executed on Fastest can be found at `http://www.flowgate.net/pdf/ftcrl.tar.gz`.

In Section 2 we introduce the TTF and in 3 we precisely state the contribution of this article. Section 4 describe the most salient features of our method. A case study is briefly introduced in Section 5. We discuss some similar approaches in Section 6 and our conclusions in Section 7.

## 2 The Test Template Framework

In this section we briefly introduce those steps of the TTF strongly related to test case refinement—for a thorough introduction see [17, 18, 3]. The presentation is made by means of an example that we will use throughout this article. It is assumed that the reader is fluent in the Z notation. In the TTF each operation within the specification is analysed to generate abstract test cases, as follows:

1. Consider the valid input space (VIS) of a Z operation.
2. Apply one or more testing tactics in order to partition the VIS.
3. Find one abstract test case from each satisfiable test condition.

We will introduce these steps for the operation named *NewClient* of the Z specification shown in Figure 2. The specification is about the savings accounts of a simple banking system. Table 1 summarizes the meaning of each basic element of the model. We think that this table plus the common knowledge about savings accounts will suffice to understand the model.

$$[AN, UID, NAME]$$
$$BALANCE == \mathbb{N}$$

State schema for the savings accounts.

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

$$\begin{array}{|l} \textit{NewClientOk} \\ \hline \Delta Bank \\ u? : UID \\ name? : NAME;\ n? : AN \\ \hline u? \notin \dom clients \\ n? \notin \dom balances \\ clients' = clients \cup \{u? \mapsto name?\} \\ balances' = balances \cup \{n? \mapsto 0\} \\ owners' = owners \cup \{u? \mapsto n?\} \\ \hline \end{array}$$

$$ClientAlreadyExists == [\Xi Bank;\ u? : UID \mid u? \in \dom clients]$$
$$AccountAlreadyExists == [\Xi Bank;\ n? : AN \mid n? \in \dom balances]$$
$$NewClient == NewClientOk \lor ClientAlreadyExists \lor AccountAlreadyExists$$

**Fig. 2.** Part of a Z specification of the savings accounts of a banking system.

| Term | Meaning |
|---|---|
| $AN$ | The set of possible savings accounts numbers |
| $UID$ | The set of identifiers of individuals |
| $NAME$ | The set of names of individuals |
| $clients\ u$ | The name of person $u$ as is recorded in the bank |
| $balances\ n$ | The balance of savings account $n$ |
| $owners(u, n)$ | $u$ is an owner of account $n$ |
| $NewClient(u, name, n)$ | Account $n$ is opened by client $u$ whose name is $name$ |

**Table 1.** Meaning of the basic elements of the Z model of Figure 2.

*Step 1.* Since *NewClient* is a total operation, its VIS is the Z schema declaring all the input and before state variables used by it:

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

*Step 2.* The TTF partitions the VIS by applying one or more *testing tactics*. The result is a set of so-called *test specifications*. 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 think that the test specifications will uncover enough errors in the implementation. Each tactic indicates how the current test specification must be partitioned by giving a set of predicates characterizing each resulting test specification. Two of the testing tactics proposed within the TTF are Disjunctive Normal Form (DNF) and Standard Partitions (SP) [3, 17].

In this example, we first apply DNF to the VIS of *NewClient*, getting the following test specifications:

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

SP is applied to the set union operator ($\cup$) in $clients \cup \{u? \mapsto name?\}$ in order to partition $NewClient_1^{DNF}$, yielding the following satisfiable test specifications (the unsatisfiable ones have been omitted for brevity):

$$NewClient_2^{SP} == [NewClient_1^{DNF} \mid clients = \varnothing \land \{u? \mapsto name?\} \neq \varnothing]$$
$$NewClient_4^{SP} == [NewClient_1^{DNF} \mid clients \neq \varnothing \land clients \cap \{u? \mapsto name?\} = \varnothing]$$

*Step 3.* The TTF prescribes to derive abstract test cases only from those test specifications that were not partitioned—we have four in the example. This means to find at least one element satisfying each of them. For example, the

following horizontal schemas represent abstract test cases of the corresponding test specifications[5]:

$$
\begin{aligned}
&NewClient_1^{ATC} == [NewClient_2^{SP} \mid balances = \varnothing \land name? = name0 \land \\
&\quad n? = an0 \land u? = uid0 \land clients = \varnothing \land owners = \varnothing] \\
&NewClient_2^{ATC} == [NewClient_4^{SP} \mid u? = uid0 \land name? = name0 \land \\
&\quad n? = an0 \land balances = \{(an1, 20)\} \land \\
&\quad clients = \{(uid1, name0)\} \land owners = \{(uid1, an1)\}] \\
&NewClient_3^{ATC} == [NewClient_2^{DNF} \mid balances = \varnothing \land name? = name0 \land \\
&\quad n? = an0 \land u? = uid0 \land clients = \{(uid0, name0)\} \land owners = \varnothing] \\
&NewClient_4^{ATC} == [NewClient_3^{DNF} \mid n? = an0 \land name? = name0 \land \\
&\quad balances = \{(an0, 0)\} \land u? = uid0 \land clients = \varnothing \land owners = \varnothing]
\end{aligned}
$$

As 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 [1, 2]. Some of these equalities specify the initial state for the test, while others specify the values for the input parameters of the SUT. This is a key issue when test case refinement is considered.

Note that test specifications and abstract test cases are all expressed in Z.

In the TTF test cases do not include test oracles because they are provided at the end of the MBT process [3]. Since oracles appear at the end of the process we do not need to deal with them during test case refinement. In the TTF, test case refinement concerns only with state and input data refinement.

## 3 A Method for Test Case Refinement

The core of this paper is, then, a general method for refining test cases, like $NewClient_4^{ATC}$, written in LaTeX markup, into executable programs or scripts written in some programming language. The result of this refinement is a collection of *concrete test cases*, or just test cases when it is clear from context. This refinement requires: (a) identifying the SUT's state variables and input parameters that correspond to the specification variables; (b) initializing the implementation variables as specified in each abstract test case; (c) initializing implementation variables used by the SUT but not considered in the specification; and (d) performing a sound refinement of the values of the abstract test cases into values for the implementation variables. For instance, if account numbers are implemented as integer numbers, then $an0$ in $NewClient_4^{ATC}$ must be refined as, say, 9711048.

The method yields programs written in the SUT's implementation language because we found it natural to correlate specification and implementation variables and it is easier to initialize them, assuming the SUT's source code is available. The correlation between specification and implementation variables is given

[5] Identifiers like $name0$ are assumed to be declared in axiomatic definitions and are regarded as constants of their types.

by engineers by means of so-called *refinement rules*, written in a declarative TCRL which is, in principle, independent of any programming language.

In summary, the method receives a user-defined refinement rule for a given Z operation, a list of test cases for that operation and the name of a programming language, and automatically applies the refinement rule to the list of test cases outputting a list of concrete test cases written in that programming language, each of which:

1. Sets the initial state of the SUT as specified by the test case.
2. Sets the input parameters expected by the SUT as specified by the test case.
3. Calls the SUT.

As it can be seen, the method we propose can be thought as a lightweight form of what is traditionally called *data refinement* [22, chapter 10]. Furthermore, as we have anticipated in the introduction, this method does not assume that the SUT was formally developed because no information from a possible formal refinement is needed.

All the remaining test activities—i.e. compiling test cases, executing them, capturing their output, etc. (Figure 1)—are beyond the scope of this paper.

## 4 Fastest Test Case Refinement Language

The method we propose is called Fastest TCRL (FTCRL). FTCRL is an interpreted language whose programs are refinement rules. Refinement rules transform a list of abstract test cases generated by Fastest into a list of concrete test cases in the SUT's programming language. The interpreter receives the target programming language as a parameter. In this paper we show part of the FTCRL semantics when the target programming language is C [23]; in [21] the full operational semantics for C and Java can be found.

The TTF is intended to be used as an MBT method for unit testing. Therefore, given a unit of implementation, $\mathsf{P}$, engineers must find the Z schema, $S$, that specifies $\mathsf{P}$—this schema may reference other schemas and it can be the specification of other units as well. Then, a refinement rule for the pair $(\mathsf{P}, S)$ must be given.

### 4.1 An Example of a Refinement Rule

Since refinement rules are essentially specifications of how VIS variables must be refined into implementation variables, we need some information about the unit under test (UUT). Below we introduce a typical refinement rule that is explained and analysed in the following sections.

Assume the banking system specified in Section 2 is implemented in the C programming language[6]. Let's say that elements of $AN$ and $NAME$ are implemented as character strings, elements of $UID$ are integer numbers and those

[6] We assume the reader is familiar with the C programming language [23].

of *BALANCE* are floats. Say *clients* is implemented as a simply-linked list, `c`, declared as:

```
struct cdata {int uid; char *name; struct cdata *n;} *c;
```

*balances* is implemented as an array, `b`, declared as:

```
struct bdata {char* num; float bal;} b[100];
```

and there is an integer variable, `l`, pointing to the last used component of `b`. *owners* is implemented as a doubly-linked list, `o`, declared as:

```
struct odata {int *puid; char *pn; struct odata *n,*p;} *o;
```

where `puid` should point to the `uid` member of the corresponding node in `c`; `pn` should point to the `num` member of the corresponding `b` component; and `n` and `p` are pointers to the next and previous nodes in the list, respectively. Say that `c`, `b`, `l` and `o` are global variables. Finally, let's assume that *NewClient* is implemented by a C function with the following signature:

```
int newClient(int u, char *name, char *n)
```

Figure 3 shows the refinement rule for *NewClient*, when it is implemented by `newClient()` and the data structures described above. Figure 4 shows the concrete test case generated by applying that refinement rule to $NewClient_2^{ATC}$. Note: (a) the kind of information of the UUT that is needed to write a refinement rule; (b) FTCRL assumes that the SUT's source code is available; and (c) Figure 4 is an executable C program. Please, look at these figures while we introduce FTCRL below.

### 4.2 The Basic Structure of a Refinement Rule

The first line in a refinement rule declares its name. Refinement rules have four mandatory sections that must be written in strict order: `@PREAMBLE`, `@LAWS`, `@UUT` and `@EPILOGUE`. The interpreter uses the preamble to collect typing information of the UUT and adds it at the beginning of a test case. The preamble should contain all the code necessary to compile the UUT—for instance, UUT's definition, type declarations, sentences to import external resources, header files, etc. The epilogue should contain code to perform clean-up once the test has been run—for instance, deleting a file—and it is blindly copied at the end of each test case. The `@UUT` section contains only one line of FTCRL code to call the UUT. The value returned by the UUT is not considered since it does not affect refinement, but other steps of the MBT process.

The name of a refinement rule can be used in other refinement rules as shown in Figure 5, with the obvious meaning. Note that this mechanism allows users to use the same `@LAWS` section with different preambles and epilogues, thus making it possible to refine the same abstract test cases to different programming languages, since all the code of the refinement rule that depends on the target programming language is confined to these two sections. The language includes others forms of reuse [21].

```
@RRULE bank
@PREAMBLE
#include <bank.h>
@LAWS
l1:u?       ==> u
l2:name?    ==> name
l3:n?       ==> n
l4:clients  ==> c AS LIST[SLL,n] WITH[clients.@dom ==> c.uid,
                                      clients.@ran ==> c.name]
l5:balances ==> b AS ARRAY WITH[balances.@dom ==> b.num,
                                balances.@ran ==> b.bal];
                balances.@# ==> l
l6:owners   ==> o AS LIST[DLL,n,p]
                     WITH[owners.@dom ==> o.puid AS REF[c.uid],
                          owners.@ran ==> o.pn AS REF[b.num]]
@UUT newClient(u,name,n)
```

**Fig. 3.** Refinement rule for *NewClient*. `bank.h` declares all the elements of the UUT.

```
#include <bank.h>
int main() {
 int u = 345;
 char *name = "name0", *n = "an0";
 struct cdata cdata0 = {87,"name0",NULL};
 struct bdata bdata0 = {"an1",20};
 struct odata odata0 = {0,0,NULL,NULL};
 c = &cdata0;
 b[0] = bdata0;
 l = 1;
 odata0.puid = &cdata0.uid;
 odata0.pn = bdata0.num;
 o = &odata0;
 newClient(u,name,n);
 return 1;
}
```

**Fig. 4.** Concrete test case for $NewClient_2^{ATC}$ generated by `bank` of Figure 3.

```
@RRULE otherBankingRefRule
@PREAMBLE bank.@PREAMBLE
@LAWS
bank.l04
.....
commercialAccounts.@LAWS
.....
@UUT deposit(....)
```

**Fig. 5.** Refinement rules can be reused as Z schemas are reused by schema inclusion.

### 4.3 Refinement Laws

The `@LAWS` section is a list of *refinement laws* (or laws), of the following form:

```
ident:list_of_spec_vars ==> refinement
```

where `ident` is an identifier to reuse the law in other rules (Figure 5), `list_of_spec_vars` is a list of one or more specification variables, and `refinement` specifies how the specification variables must be refined. The token `==>` can be read as 'refines to'.

The most simple law is, for instance, `l1` in Figure 3. For each abstract test case, this law makes the interpreter to declare a local variable named `u` of type `int` and to assign it the value of $u?$ in the abstract test case (Figure 4). The type of `u` is deduced as follows: `u` is the first parameter in the call placed in the `@UUT` section, and the first parameter found in the signature of `newClient()` is of type `int`. In general, all the typing information can be deduced by parsing both the LaTeX markup of the Z specification and the source code of the SUT. Constant values of given types at the specification level, such as $uid0$, are translated to the implementation type by applying an arbitrary bijection whenever necessary.

Note that, in this context, the overflow C semantics of the `int` type is not a problem when refining Z's $\mathbb{Z}$, because, if at the Z level a natural number is greater than the C `int` limit, then, precisely, this test case will test how the program deals with the overflow C semantics. It is not the difficulty appearing in classical refinement calculus: the intention is to test the program, not to refine the specification.

Law `l4` specifies that *clients* is implemented as the `c` list. The first parameter of the `LIST` clause indicates that `c` is a simply-linked list and the second one is the name of the variable pointing to the next node in the list—some of these parameters are ignored when refining to some programming languages, Java is an instance. It is necessary to include this information in the law because, in some programming languages, it is impossible to automatically deduce that `c` is a list, solely from its declaration. The `WITH` clause helps to specify how each ordered pair in *clients* must be accommodated in the list. In this case, elements in the domain go to `uid` and elements in the range to `name`. Therefore, the interpreter creates a new variable of type `cdata` for each pair in *clients* and initializes them with the constant values of each pair. The value of the `n` member of each of these new variables is set to point to the address of any other of them—since *clients* is a function, there is no order between its pairs, and so any order in `c` should be correct. In general, FTCRL applies a sort of extensionality to refine Z sets [21].

Note how a specification variable is refined to more than one implementation variable in `l5`; `balances.@#` is the cardinality of *balances*. Had it been necessary to make `l` to point to the first free component in `b`, then we would have written: `balances.@# + 1`—in general, any constant expression is valid.

Regarding `l6`, `DLL` stands for doubly-linked list and the other two parameters are the members pointing to the next and previous nodes, respectively. If an implementation variable is intended to hold a reference (or a pointer) to some data in some other data structure, the `REF` directive must be used. It is possible

to generate source code according to this specification because every element of a dynamic data structure is first saved in a new static variable whose name, memory address and value can be freely used by the interpreter.

### 4.4 More Examples and Features

In this section we will show a few small examples to introduce a variety of FTCRL's features; sometimes we will use the savings account example.

*Two specification variables refined into one implementation variable.* Consider the following excerpt from some specification:

$[NAME]$
$AddPerson == [first?, last? : NAME \ldots \mid \ldots]$

Assume the implementation stores the first and last name of persons in a single character string variable, `name`. Then, the law could be as follows:

```
person:first?, last? ==> last? ++ ", " ++ first? ==> name
```

If an abstract test case binds *name*0 to *first*? and *name*1 to *last*?, then the interpreter would generate the following code:

```
char* name = "name1, name0";
```

*Implementation details abstracted away in the specification.* Now assume the implementation of the banking system introduced in Section 2 stores also the address and age of each client. Specifiers abstracted away these details retaining only the name of the client. Therefore, `cdata` would indeed be:

```
struct cdata {int uid, age; char *name, *addr; struct cdata *n;} c;
```

In this case the refinement law would be:

```
l04:clients ==> c AS LIST [SLL,n]
                     WITH [clients.@dom ==> c.uid,
                           clients.@ran ==> c.name,
                           "Road" ==> c.addr,
                           40 ==> c.age]
```

or `@AUTOFILL ==> c.*` can replace `"Road" ==> c.addr, 40 ==> c.age` [21]. In other words, if an implementation detail was abstracted away in the specification, then, in some way, its value is irrelevant with respect to the correctness of the implementation. Hence, the same value can be used in all of the tests.

*Refining into external resources.* Assume there is an UUT of the banking system that reads client data from a text file. Test cases for this UUT would need to initialize this file according to the value *clients* has in different abstract test cases. Say the file stores one record per line with the format `UID:NAME`. Then, the refinement law would be:

```
file:clients ==> clients.@DOM ++ ":" ++ clients.@RAN
                 ==> clientData.txt AS FILE[/bank]
```

If in some test case we have $clients = \{(uid1, name0), (uid2, name1)\}$ the interpreter would produce roughly the following C code:

```
fd = open(/bank/clientData.txt, O_RDWR | O_TRUNC | O_CREAT);
.......
write(fd, "87:name0", strlen("87:name0"));
write(fd, "91:name1", strlen("91:name1"));
.......
close(fd);
```

where 87 and 91 result from applying an arbitrary bijection between *UID* and `int` as we have said before.

*Refining complex Z types.* Suppose it is necessary to refine $f : X \pfun Y \pfun H \times W$ where $X$, $Y$ and $W$ are given types and $H$ is the schema $[a : A;\ b : B]$. The recursive nature of FTCRL, Z and all programming languages make it possible to refine such complex types in equally complex implementation data structures. For instance, the dot notation in FTCRL can be recursively applied to cross products, schema types and other constructions [21].

*Data structures currently supported.* The implementation data structures that are supported by FTCRL depends on the programming language. For C and Java we have [21]:

- C: `int` (plus all the modifiers `short`, `long`, `unsigned` and `signed`), `char`, `float`, `double`, `enum`, arrays, `struct` and pointers to any of them. This implies that all kinds of lists are supported.
- Java: `int`, `short`, `long`, `byte`, `Integer`, `Short`, `Long`, `Byte`, `char`, `Character`, `float`, `double`, `Float`, `Double`, `enum`, arrays, `class`, `List<type>`, `ArrayList<type>`, `LinkedList<type>`, `Attributes`, `HasMap`, `Hashtable`, `IdentityHashMap`, `TreeMap`, `WeakHashMap` and `String`.

*Completeness—Refining to possibly unsupported data structures.* Say some C program defines a list where each node points to the next node but also to the node five positions ahead. Data structures like this can be arbitrary complex, but, as far as we know, they are seldom used. FTCRL was designed to directly support the most common data structures, but it provides a (low level) language feature that allows to refine to any data structure. This feature involves using

the `@PLCODE` optional section. This section can contain only source code (of the SUT's programming language) and is blindly copied between the code generated after parsing `@LAWS` and the call to the UUT. We expect that users will use this section only when they find no other way of writing their refinement rules, because it increases the dependency of refinement rules on the SUT's programming language. Readers can find more about `@PLCODE` in [21, Section 2.4].

Even considering only the most common data structures, it is very difficult to prove that FTCRL can be used to refine any Z variable into any implementation data structure because it would require to prove that for every programming language. However, since FTCRL supports all the C data structures, we have strong reasons to believe that data structures defined by higher-level programming languages can be supported too. The `@PLCODE` mechanism provides completeness where the proper FTCRL code fails to do so.

*Implementation independence.* We want to emphasize that refinement rules and all the test cases generated by them are resilient to a number of changes in the implementation. For instance, considering the savings accounts example, if there is some error in updating or walking `c`, or some error in keeping the references of `o`'s nodes, or `l` is not correctly synchronized with the last used component of `b`, and these errors are fixed, the `bank` refinement rule remains the same since `c`, `o`, `b` and `l` all maintain their attributes and roles in the implementation.

*Fastest's architecture for test case refinement.* Fastest is a Java application, so it is FTCRL's interpreter. Currently, the interpreter is a proof of concept implemented with the ANTLR parser generator [24]—and using a simpler version of FTCRL than the one shown here. The architecture of the interpreter was envisioned to allow for easily plug-in modules implementing FTCRL for new programming languages, as shown in Figure 6. Some of the pluggable modules hide a few technological issues such as connections to databases, operating system interactions, etc.

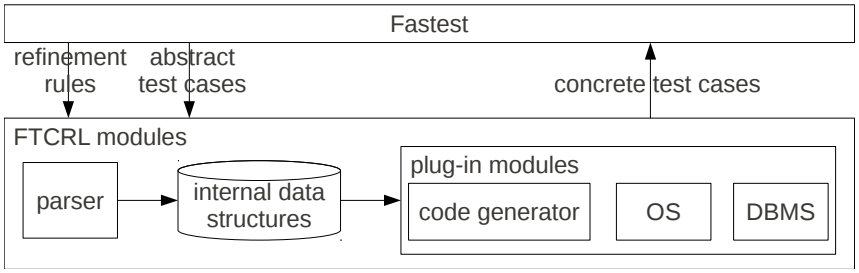


**Fig. 6.** Simple module diagram of the Fastest's architecture for test case refinement.

## 5 A Case Study

This approach has been used in a contract with Nemo Group (Argentina) to test its core product. Confidentiality issues and space restrictions impede us to include all the information; key data is available at `http://www.flowgate.net/pdf/cacheflight.tar.gz`. Nemo's core business is software development for the travel industry. The SUT is a large Java application whose purpose is to provide booking functionality for flights provided by several major international companies. This program heavily uses a database.

We have written Z specifications for the most critical methods of the key classes of the SUT. The choice of methods and classes as well as the specification for each of them had to be reverse-engineered along with some key Nemo's engineers. This process was carried out in such a way that we did not read the code. First, we asked Nemo's engineers what a particular method should do, then we wrote the specification according to their comments—how they learned the function of a method was transparent for us. Once the specifications were ready we applied Fastest to generate abstract test cases and, at the same time, we wrote the refinement rules—during this activity we seldom needed the assistance of Nemo's personnel since we have already learned the application. Currently, we have refined more than one thousand test cases with a few refinement rules. Refinement rules include database connections, nested classes, lists, etc. However, we cannot give figures about how many errors were found because the experiment concluded when we were able to execute the test cases—checking whether test cases find errors or not is the last step of the process (Figure 1) which is not fully available in Fastest, yet.

## 6 Related Work

Refinement calculus or specification refinement has a long and well-established tradition in the formal methods community [25, 26]. The Z formal notation is not an exception [22]. However, these theories are aimed at a much harder and general problem than ours: to formally transform an abstract specification into executable code. Usually, these methods list a set of sound refinement rules guaranteeing that every time they are applied, the description so obtained verifies the original specification. Classical refinement has four important differences with our method: (a) we do not try to refine the whole specification but just some constant values of some variables; (b) the implementation is already available, it must not be derived from the specification; (c) we do not attempt to prove that refinement rules are right, precisely, we try to surface problems in the implementation; and (d) we propose that users write refinement rules instead of choosing them from a fixed menu, because implementations can be arbitrarily complex. However, our approach was inspired by the idea of tiered specifications proposed for Larch [27] which can be seen as a form of refinement.

The creators of the TTF applied it to Object-Z to test classes of object-oriented programs [28]. They use the ClassBench testing framework which requires testers to write testgraphs to test the class under test. Once testgraphs

are written ClassBench automatically tests the class. The authors propose to generate a finite state machine (FSM) from a test specification and then to transform the FSM to a testgraph. However, it is not clear how easy it might be to semi-automatically derive testgraphs from abstract test cases. Actually, the authors discuss several issues that arise when transforming a FSM to a testgraph because they are models at different levels of abstraction.

Derrick and Boiten [20] analyse the relationship between testing and refinement in the context of Z specifications. However, they apply a different approach because they assume that the implementation has been refined from the specification. Therefore, they first derive abstract test cases from the Z specification—in doing so they apply a different method, not the TTF—and then they use information available in the refinement in order to refine the abstract test cases. Although their method is more formal than FTCRL, it is less applicable than ours since formal refinement is seldom available.

BTT is a MBT method based on the B notation that generates sequences of operation invocations at an abstract level that constitute the abstract test cases [5]. This sequences are made executable by translating them into scripts [29]. These scripts are built by providing a test script pattern and a mapping table. The test script pattern is a source code file in the target language with some tags indicating where to insert sequences of operation invocations. The information present in a mapping table is similar to that of a refinement rule. However, the mapping tables do not seem to be as expressive as FTCRL. Furthermore, in this method testers must provide the test script pattern instead of getting it automatically from the reification information.

AspecT is an aspect-oriented language for the instantiation of abstract test cases [30]. It starts from test cases generated from UML statecharts. This approach uses a combination of languages, Ecore, OCL, Phyton, Groovy and AspectT, to refine test cases. It does not seem to clearly define the mapping between specification and implementation variables but to decompose the refinement phase into several steps in which aspects, pointcuts and advices are written.

If some naming conventions are applied and the implementation is conveniently annotated, it might be possible to automatically define many refinement rules. Meyer and et al. manage to automatically test programs by annotating them with contracts written in the implementation language, Eiffel in this case [31]. They, for instance, use the same names for variables in the implementation and in the contracts. We need to further investigate whether this can be applied to Z specifications since they are more abstract than contracts.

## 7 Conclusions

We have proposed FTCRL, a declarative refinement language that automates test case concretization within the Test Template Framework (TTF), a Z-based MBT method. By defining simple refinement rules, that are independent of test cases and, to a great extent, of the implementation itself, testers can use an interpreter to refine all the abstract test cases generated by Fastest—TTF's

implementation. A prototype of this interpreter has been implemented in Fastest by following an architecture that allows developers to plug-in modules supporting different implementation languages.

Refinement rules become, also, a key formal document linking the specification and the implementation. It must be noted, however, that the mere possibility of writing a refinement rule does not necessarily imply that the implementation verifies the specification. Once the implementation has passed all the tests, it can be assumed correct (modulo testing) and, then, refinement rules might be used to perform some lightweight formal analyses.

We plan to improve the interpreter and to add more features to FTCRL. So far, the method is non-intrusive, i.e. it does not modify the SUT to test it—even if it is implemented in Java where reflection is used to access private members from the outside. This property is important since modifying the SUT to get it tested can be a source of artificial errors. However, we have a problem with local static variables declared inside a subroutine since they cannot be initialized from the outside of the unit under test. We need to further investigate this issue.

## References


1. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2006)
2. Hierons, R.M., et al.: Using formal specifications to support testing. ACM Comput. Surv. **41**(2) (2009) 1–76
3. Stocks, P., Carrington, D.: A Framework for Specification-Based Testing. IEEE Transactions on Software Engineering **22**(11) (November 1996) 777–793
4. Grieskamp, W., Gurevich, Y., Schulte, W., Veanes, M.: 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) 112–122
5. Legeard, B., Peureux, F., Utting, M.: A Comparison of the BTT and TTF Test-Generation Methods. In: ZB '02: Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B, London, UK, Springer-Verlag (2002) 309–329
6. Bernot, G., Gaudel, M.C., Marre, B.: Software testing based on formal specifications: a theory and a tool. Softw. Eng. J. **6**(6) (1991) 387–405
7. Posey, B.: Just Enough Software Test Automation. Prentice Hall PTR, Upper Saddle River, NJ, USA (2002)
8. Fewster, M., Graham, D.: Software test automation: effective use of test execution tools. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA (1999)
9. ISO: Information Technology – Z Formal Specification Notation – Syntax, Type System and Semantics. Technical Report ISO/IEC 13568, International Organization for Standardization (2002)
10. Ammann, P., Offutt, J.: Using formal methods to derive test frames in category-partition testing. In: Compass'94: 9th Annual Conference on Computer Assurance, Gaithersburg, MD, National Institute of Standards and Technology (1994) 69–80
11. Hall, P.A.V.: Towards testing with respect to formal specification. In: Proc. Second IEE/BCS Conference on Software Engineering. Number 290 in Conference Publication, IEE/BCS (July 1988) 159–163

12. Hierons, R.M., Sadeghipour, S., Singh, H.: Testing a system specified using Statecharts and Z. Information and Software Technology **43**(2) (February 2001) 137–149
13. Hierons, R.M.: Testing from a Z specification. Software Testing, Verification & Reliability **7** (1997) 19–33
14. Hörcher, H.M., Peleska, J.: Using Formal Specifications to Support Software Testing. Software Quality Journal **4** (1995) 309–327
15. Stocks, P.: Applying Formal Methods to Software Testing. PhD thesis, Department of Computer Science, University of Queensland (1993)
16. Maccoll, I., Carrington, D.: Extending the Test Template Framework. In: Proceedings of the Third Northern Formal Methods Workshop. (1998)
17. Cristiá, M., Rodríguez Monetti, P.: Implementing and applying the Stocks-Carrington framework for model-based testing. In Breitman, K., Cavalcanti, A., eds.: ICFEM. Volume 5885 of Lecture Notes in Computer Science., Springer (2009) 167–185
18. Cristiá, M., Albertengo, P., Rodríguez Monetti, P.: Pruning testing trees in the Test Template Framework by detecting mathematical contradictions. In Fiadeiro, J.L., Gnesi, S., eds.: SEFM, IEEE Computer Society (2010) 268–277
19. Cristiá, M., Albertengo, P., Rodríguez Monetti, P.: Fastest: a model-based testing tool for the Z notation. In Mazzanti, F., Trentani, G., eds.: PTD-SEFM, Consiglio Nazionale della Ricerche, Pisa, Italy (2010) 3–8
20. Derrick, J., Boiten, E.: Testing refinements of state-based formal specifications. Software Testing, Verification and Reliability (9) (July 1999) 27–50
21. Cristiá, M., Rodríguez Monetti, P., Albertengo, P.: The FTCRL reference guide. Technical report, Flowgate Consulting (2010)
22. Potter, B., Till, D., Sinclair, J.: An introduction to formal specification and Z. Prentice Hall PTR Upper Saddle River, NJ, USA (1996)
23. Kernighan, B.W., Ritchie, D.M.: The C Programming Language Second Edition. Prentice-Hall, Inc. (1988)
24. Parr, T.: Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages. 1st edn. Pragmatic Bookshelf (2009)
25. Morgan, C.: Programming from specifications (2nd ed.). Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK (1994)
26. Back, R.J., Wright, J.V.: Refinement Calculus: A Systematic Introduction. 1st edn. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1998)
27. Guttag, J.V., Horning, J.J.: Larch: languages and tools for formal specification. Springer-Verlag New York, Inc., New York, NY, USA (1993)
28. Carrington, D.A., MacColl, I., McDonald, J., Murray, L., Strooper, P.A.: From object-z specifications to classbench test suites. Softw. Test., Verif. Reliab. **10**(2) (2000) 111–137
29. Bouquet, F., Legeard, B.: Reification of executable test scripts in formal specicifation-based test generation: The Java card transaction mechanism case study. In Araki, K., Gnesi, S., Mandrioli, D., eds.: FME. Volume 2805 of Lecture Notes in Computer Science., Springer (2003) 778–795
30. Benz, S.: Aspectt: aspect-oriented test case instantiation. In: Proceedings of the 7th international conference on Aspect-oriented software development. AOSD '08, ACM (2008) 1–12
31. Meyer, B., Fiva, A., Ciupa, I., Leitner, A., Wei, Y., Stapf, E.: Programs that test themselves. Computer **42** (September 2009) 46–55