Representing Parnas' Uses Relation in Z for the Test Template Framework

Maximiliano Cristiá[1], Joaquín Mesuro[1], and Claudia Frydman[2]

[1] CIFASIS and UNR, Rosario, Argentina
cristia@cifasis-conicet.gov.ar, joaquin.mesuro@gmail.com
[2] LSIS-CIFASIS, Marseille, France
claudia.frydman@lsis.org

Abstract. The Test Template Framework (TTF) is a model-based testing method for the Z notation, originally proposed for unit testing. In this paper we analyze how the TTF can be extended to integration testing. Since integration testing is related to software design, we decided to investigate the relation between the TTF and the *uses* relation, a key element of David Parnas' design theory. We propose how a Z specification should be structured for the TTF to be able to generate integration tests by following the *uses* relation. The problem of stub generation and the kinds of errors that these integration tests can discover are also discussed.

1 Introduction

The Test Template Framework (TTF) is a model-based testing (MBT) method proposed for the Z notation [26]. In the TTF each Z operation schema is analyzed to generate (abstract) test cases. Each operation schema in a Z model is the specification of a piece of code in the implementation that sometimes corresponds to a unit of implementation. This is why we say that the TTF generates unit tests. Recently the TTF was automated roughly to the same degree of other MBT methods by a tool called Fastest [8]. This makes the TTF and Fastest appealing options for unit testing within the Z community.

According to the accepted practice of Software Engineering, after each unit of implementation has been tested in isolation, they should be incrementally integrated and tested [15, 25]. This phase or level of testing is known as integration testing. On the other hand, software design is defined as the decomposition of a system into software elements, the description of what each element is intended to do and the relations among these elements [15]. Therefore, integration testing is influenced by the design. Furthermore, the design and the structure of the functional specification influence a MBT method when is applied during integration testing because test cases are derived from the specification and executed on the elements of the design. On the other hand, if software elements are related to each other then errors in one of them may cause errors in the others. The accepted solution is to build so-called stubs units which mimic the behavior of the real units but only for a few inputs. Manually crafting such stubs is a source of costs and errors. Building the minimum number of correct stubs avoiding as much manual work as possible can be considered as the *stub generation problem.* Stubs are necessary because, in general, units are tested correct (and not proven correct), so these units cannot be used while other units are tested (because the formers can induce errors in the latter).

The contributions of this paper are the following: a) a set of guidelines for writing Z specifications that will simplify (TTF-based) test case generation during integration testing; b) an integration strategy based on Parnas' *uses* relation that reduces the number of manually crafted stubs; c) a set of conditions that guarantee that a unit can be used as stub of itself without inducing errors (in other units) during integration testing; and d) an analysis of the types of (integration) errors this method can discover.

An example motivatiing the issues discussed in this paper is given in Sect. 2. After introducing the TTF in Sect. 3, three main problems are addressed: a) how a Z specification should be structured and linked with the design to best serve for integration testing, in Sect. 4; b) what is the best strategy in the TTF to incrementally integrate units so integration testing can benefit from unit testing, in Sect. 5; and c) the stub generation problem, in Sect. 6. The kinds of errors that the extended TTF can find are analyzed in Sect. 7. All the results obtained in these sections are discussed in Sect. 8. A comparison with similar approaches can be found in Sect. 9 and our conclusions in Sect. 10.

In this paper "unit" means "subroutine" which in turn includes "function", "procedure" and "method". Our work aims at integrating units for which the source code is available. All the units that are integrated belong to the same executable but can belong to different modules. This work does not make any assumptions about the implementation technology. A general theory of software design and first-order logic over a set theory (i.e. Z) are the fundamentals.

This paper is a summary of an unpublished paper available on-line [11]. We assume the reader is familiar with the Z notation.

2 Motivation

In this section we want to show some of the issues that MBT faces when integration testing is considered. We will do it by means of a simple example. Assume we need to implement the following functionality: receive an integer number, check whether it belongs to a list and, if it does not, then add it to the list and sort the list. A possible Z specification for this requirements is as follows.

$$S \mathrel{\widehat=} [\mathit{list} : \operatorname{seq} \mathbb{Z}]$$
$$\mathit{InList} \mathrel{\widehat=} [\Xi S;\ x? : \mathbb{Z} \mid x? \in \operatorname{ran} \mathit{list}]$$

$$\begin{aligned}
&\mathit{InsertOk} \mathrel{\widehat=}\\
&\quad [\Delta S;\ x? : \mathbb{Z} \mid x? \notin \operatorname{ran} \mathit{list} \land \operatorname{ran} \mathit{list}' = \operatorname{ran} \mathit{list} \cup \{x?\}\\
&\qquad\qquad \land (\forall i, j : \operatorname{dom} \mathit{list}' \bullet i < j \Rightarrow \mathit{list}'\, i < \mathit{list}'\, j)]\\
&\mathit{InsertAndSort} \mathrel{\widehat=} \mathit{InsertOk} \lor \mathit{InList}
\end{aligned}$$

Clearly, a first design could be to implement *InsertAndSort* with a single subroutine. However, since *InsertAndSort* includes the specification of a sorting algorithm, it is reasonable to decompose (i.e. design) its implementation into two subroutines, insert and sort, with the following functionality: insert reads the element to be inserted, checks whether it belongs to the list and, if not, calls sort, which inserts the element in the list and sorts it[1]. Then, a possible Z specification reflecting this design is as follows.

$$\begin{array}{|l}
\textit{Sort}_1 \\
\hline
\Delta S \\
x? : \mathbb{Z} \\
\hline
\# \textit{list}' = \# \textit{list} + 1 \\
\textit{list}' \upharpoonright \{x?\} = (\textit{list} \upharpoonright \{x?\}) \frown \langle x? \rangle \\
\forall\, y : (\operatorname{ran} \textit{list}) \setminus \{x?\} \bullet \textit{list} \upharpoonright \{y\} = \textit{list}' \upharpoonright \{y\} \\
\forall\, i, j : \operatorname{dom} \textit{list}' \bullet i < j \Rightarrow \textit{list}'\, i \leq \textit{list}'\, j \\
\hline
\end{array}$$

$$\textit{Input}_1 \mathrel{\widehat{=}} [\Delta S;\ x? : \mathbb{Z} \mid x? \notin \operatorname{ran} \textit{list}]$$
$$\textit{InsertAndSort}_1 \mathrel{\widehat{=}} (\textit{Input}_1 \land \textit{Sort}_1) \lor \textit{InList}$$

where S and *InList* are the same as above. In this case, sort implements $\textit{Sort}_1$ whereas insert implements $\textit{InsertAndSort}_1$ replacing $\textit{Sort}_1$ by a call to sort. $\textit{Sort}_1$ is more complex than *InsertOk* because it can sort lists with or without duplicates. In other words, $\textit{Sort}_1$ is more general than *InsertOk*, although in our example it is called only when a new element is to be inserted.

In summary, we have two designs with different subroutines for the same requirements. Given that the specification of each unit is different in each design, test cases generated by a MBT method should be different when applied to each design. Furthermore, the specification is saying that the correctness of insert depends upon the correctness of sort because $\textit{Sort}_1$ is part of $\textit{InsertAndSort}_1$. This dependency should impose an order for testing these units that should be taken into account by the MBT method. If, on the contrary, insert is tested before sort has passed all of its test, errors in insert may be difficult to track down because they may come from itself or from sort. However, even if sort has passed all of its test, an error found while insert is tested cannot be blamed just to itself because sort has not been proven correct, it was just tested. Then, we either build a (correct) stub of sort for testing insert, or we prove that test cases run on insert always call sort as it was called when it was tested—and since it passed all its test then errors found while insert is tested can be blamed just to itself. Finally, if, for instance, insert calls an error reporting routine, say err, when $x? \in \operatorname{ran} \textit{list}$, should err be tested before insert? We believe it should not necessarily be the case because err is not part of insert's specification (i.e. $\textit{InsertAndSort}_1$). In other words, the correctness of insert does not depend on

[1] In this paper, we use *math* text to represent the specification of subroutines written in sans serif. For example, $\textit{InsertAndSort}_1$ is the specification of insert.

err. This implies, in turn, that any stub of err will do during insert's testing. In summary, a convenient adaptation of a MBT method can help in many ways during integration testing as we will show in the rest of this paper.

3 Introduction to the TTF and Fastest

In this section we present just the main concepts of the TTF and Fastest; for deeper presentations consult [26, 8, 12, 9, 10]. Fastest generates test cases for each operation schema selected by the user in a Z model. If A is an operation then its valid input space (VIS) is defined as the following Z schema:

$$A^{VIS} \mathrel{\widehat=} [x_1 : X_1;\ \dots;\ x_n : X_n \mid \mathrm{pre}\ A]$$

where $x_1 : X_1;\ \dots;\ x_n : X_n$ are all the input and state variables declared in A after full schema expansion, and pre A is the precondition of A.

The goal of the TTF is to partition A^{VIS} by applying so-called testing tactics. A testing tactic is a systematic way of dividing the *VIS* of a Z operation. Some tactics are: disjunctive normal form (DNF), standard partitions (SP), free types (FT), etc. [8, 10]. After a testing tactic is applied to A^{VIS} a family of test conditions[2] is obtained. These test conditions usually form a partition of the *VIS*. In Fastest they are formalized as Z schemas as follows:

$$A_1^{T_1} \mathrel{\widehat=} [A^{VIS} \mid P_1^{T_1}(x_1, \dots, x_n)] \quad \dots \quad A_{m_1}^{T_1} \mathrel{\widehat=} [A^{VIS} \mid P_{m_1}^{T_1}(x_1, \dots, x_n)]$$

where T_1 is the name of the tactic and $P_i^{T_1}(x_1, \dots, x_n)$ for $i \in 1 \mathrel{..} m_1$ are predicates generated by T_1. These predicates are called characteristic predicates of the test conditions. $P_i^{T_1}$ defines the conditions for a test case. In other words, a test condition is a set of test cases satisfying a given condition or predicate.

Perhaps the most important feature of the TTF is that it proposes to apply other tactics to one or more of the test conditions already generated, thereby getting progressively more detailed test conditions. For example, if testing tactic T_2 is applied to $A_1^{T_1}$ the following test conditions are generated:

$$A_1^{T_2} \mathrel{\widehat=} [A_1^{T_1} \mid P_1^{T_2}(x_1, \dots, x_n)] \quad \dots \quad A_{m_2}^{T_2} \mathrel{\widehat=} [A_1^{T_1} \mid P_{m_2}^{T_2}(x_1, \dots, x_n)]$$

Observe how schema inclusion is used to link test conditions between them and with the *VIS*. Note, also, that schema inclusion adds more predicates to a test condition. In effect, if $A_1^{T_1}$ is expanded, for instance, inside $A_2^{T_2}$ we have:

$$A_2^{T_2} \mathrel{\widehat=} [A^{VIS} \mid P_1^{T_1}(x_1, \dots, x_n) \land P_2^{T_2}(x_1, \dots, x_n)]$$

Schema inclusion organizes test conditions in a so-called testing tree which has the *VIS* in the root, the first test conditions in the first level, and so forth.

In the TTF a test case is a Z schema where each variable declared in the *VIS* is equal to a constant value such that the predicate of the corresponding leaf is satisfied. For example, a test case for $A_2^{T_1}$ is:

$$A_2^{TC} \mathrel{\widehat=} [A_2^{T_1} \mid x_1 = c_1 \land \dots \land x_n = c_n]$$

[2] Also called test templates, test specifications, test classes, etc.

4 Structuring a Z Specification for Integration Testing

As we have said in the introduction, integration testing is strongly related to software design. The approach to integration testing based on a MBT method proposed in this paper is based on what David L. Parnas calls "uses relation" or "uses structure" [23], a key concept of his seminal work on software design. The *uses* relation is a binary relation between subroutines. If P and Q are two subroutines, then $\mathsf{P}\ \mathit{uses}\ \mathsf{Q}$ if "there exist situations in which the correct functioning of P depends upon the availability of a correct implementation of Q" [23]. Note that the *uses* relation differs from the *calls* (or *invokes*) relation[3] because: (a) if P's specification requires only that $\mathsf{P}\ \mathit{calls}\ \mathsf{Q}$ then it is enough for P to call Q when its specification says so, from P's perspective Q can be correct or not; and (b) P may use Q by sharing some data structures although the former never calls the latter. According to Parnas, "the design of the *uses* hierarchy should be one of the major milestones in a design effort".

The *uses* relation is relevant to MBT methods since it is based on the specification of a subroutine. In effect, $\mathsf{P}\ \mathit{uses}\ \mathsf{Q}$ means that the *specification* of P says that it needs a correct version of Q. From a functional perspective P and Q could be implemented in a single unit whose specification is, roughly, the conjunction of P's and Q's specifications. However, from a design perspective it is better to split this unit into two in such a way that one uses the other. We have shown an example of this situation in Sect. 2. Given that the TTF uses Z specifications, it is worth to study how to write them so it is easy to find the *uses* relation.

We propose the following guidelines for writing Z specifications that will be used during integration testing.

– Each subroutine is specified by a schema. More precisely, for each subroutine P there must be a named schema A which is its specification.
– Users must generate test cases only for those schemas that are the specifications of subroutines. For example, users must generate test cases for $\mathit{InsertAndSort}_1$ and Sort_1 but not for Input_1 and InList. In fact, test cases covering the functionality specified in Input_1 and InList will be generated as part of the test cases generated from $\mathit{InsertAndSort}_1$ [8].
– Let A and B be Z schemas describing the specification of subroutines P and Q, respectively[4]. If $\mathsf{P}\ \mathit{uses}\ \mathsf{Q}$ and $\mathsf{P}\ \mathit{calls}\ \mathsf{Q}$, then A must be written as follows:

$$A \mathrel{\widehat=} \mathcal{SE}(B, A_1, \dots, A_n) \qquad (\dagger)$$

where $\mathcal{SE}$ is some schema expression depending on schemas B and $A_1, \dots, A_n$. That is, Q's specification is part of P's which is completed by the A_i schemas. An example of this is $\mathit{InsertAndSort}_1$ given in Sect. 2. If P uses other subroutines besides Q, then their corresponding Z schemas will also participate

[3] $\mathsf{P}\ \mathit{calls}\ \mathsf{Q}$ includes the case where P calls Q indirectly by a chain of calls through some intermediate subroutines. *uses* is also a transitive relation.
[4] This convention is used across the paper. Recall: *math* is specification and sans serif is implementation.

in (†) like B. For the remaining of this paper we will use (†) but all the results can be extended to the more complex case where P uses more than one subroutine.
- If P *calls* Q but P $\mathit{u\not{s}es}$ Q, then B must not be part of A because P $\mathit{u\not{s}es}$ Q means that P's specification says that it does not depend on Q. So including B in A would be an error because this would indicate a functional dependency of P on Q. An example of this second scenario is when insert calls err, also discussed in Sect. 2.
- If P *uses* Q but P $\mathit{ca\not{l}ls}$ Q, then B must not be part of A, at least concerning integration testing. This case is further discussed in *Global errors* in Sect. 7.

Since the case P *uses* Q $\land$ P *calls* Q is analyzed several times in this paper, from now on we will write P $\overline{\mathit{uses}}$ Q as a shorthand for it. Furthermore, we will write $\overline{\mathit{uses}}$ as a synonym of "use and call".

Capturing the differences between the *uses* and *calls* relations in the specification has important consequences for integration testing. Assume that P *calls* Q. Then, a stub of Q will be necessary when P is unit-tested. In general, this stub should verify B (i.e. Q's specification) because otherwise P might look erroneous when, actually, the errors may come from Q's stub. Now, also assume that P $\mathit{u\not{s}es}$ Q. Then, Q's stub can be anything complying with Q's signature (even Q itself) because P's correctness does not depend on Q's. Therefore, if P *calls* Q but P $\mathit{u\not{s}es}$ Q we can conclude that when P is tested: (a) Q's stub can be automatically generated or Q can be used if it is available; and (b) if integration testing shows errors in P they cannot be due to the presence of an incorrect Q.

5 Integration Testing within the TTF

Guiding integration testing by the *uses* relation has a number of benefits. If P *uses* Q the very nature of testing impedes to restrict the search for the cause of an error exposed during the testing of P just to itself because it depends at least on Q which, at best, was already tested, but not proven correct. This is one of the greatest difficulties during integration testing as testing of subroutines who use dozens of others tend to exacerbate that problem. If integration testing is guided by the *uses* relation this problem is minimized, as we will show below.

Parnas restricts the *uses* relation to a hierarchy because otherwise "one may end up with a system in which nothing works until everything works" [23]. If *uses* is a hierarchy, there is a set of subroutines, $\mathfrak{U}_0$, which do not use other subroutines. These should be the first to be tested because the cause of an error in one of them should be located only in itself. Then, there is another set of subroutines, $\mathfrak{U}_1$, whose members only $\overline{\mathit{uses}}$ subroutines in $\mathfrak{U}_0$. These should be the second to be tested, right after those in $\mathfrak{U}_0$ have passed all of their tests. Moreover, in general, there will be a family of sets $\mathfrak{U}_1^0(i) \subseteq \mathfrak{U}_1$, for $i \in 1 \mathrel{..} \#\mathfrak{U}_0$, whose subroutines use exactly i subroutines (of $\mathfrak{U}_0$)[5]. Then, it would be better to test the subroutines of $\mathfrak{U}_1$ according to the following order: $\mathfrak{U}_1^0(1), \dots, \mathfrak{U}_1^0(\#\mathfrak{U}_0)$.

[5] In general, some of the $\mathfrak{U}_1^0(i)$ will be empty.

In this way subroutines using less subroutines are tested before those using more, which is helpful when searching for the cause of an error.

Clearly, a family of sets $\mathfrak{U}_i$, with $i \in 2 \mathrel{..} n$ for some n, whose subroutines $\overline{uses}$ one or more subroutines in $\mathfrak{U}_0 \cup \dots \cup \mathfrak{U}_{i-1}$ should be defined to organize integration testing as was just explained for $\mathfrak{U}_0$ and $\mathfrak{U}_1$. This is what we call integration testing guided by the *uses* relation. Note that all these sets can be computed automatically from the Z specification if our guidelines are followed (cf. Sect. 4). See [11] for more details, examples and formal definitions of sets $\mathfrak{U}_k$ and $\mathfrak{U}_k^j(i)$.

Test case generation during integration testing. If P and Q are going to be tested using a MBT method then their specifications, A and B, must be analyzed in order to generate their abstract test cases. The question is whether the relation $\mathsf{P}\,\overline{uses}\,\mathsf{Q}$, and thus the fact that A includes B, would change the standard way in which the MBT method is applied. If the MBT method analyses the inner details of formulas A and B then some adaptation is required because otherwise it will expand B inside A meaning that test cases generated for P will be influenced by Q as well. However, Q was already tested as a unit and has passed all of its tests, so, in principle, there is no point in considering it again. Moreover, if the transitive closure of $\overline{uses}$ includes a long chain of subroutines starting from P, then fully expanding A will result in a huge formula which will be hard to analyze by any implementation of the MBT method. This is in line with the idea that during integration testing units already tested should be treated as black boxes. On the other hand, if B is not expanded inside A it might be the case that Q is not tested as thoroughly as it would if the expansion had been performed. This point will be discussed in Sect. 7.

Adapting the TTF to Integration Testing. The TTF is applied to elements belonging to $\mathfrak{U}_0$ as it is [8]. If $\mathsf{P} \in \mathfrak{U}_1$, then $A \mathrel{\widehat=} \mathcal{SE}(B, A_1, \dots, A_n)$ for some B such that it is the specification of some $\mathsf{Q} \in \mathfrak{U}_0$. In this case, when the TTF is applied to A, B is not fully expanded, contradicting the original presentation of both the TTF and Fastest. Only variables declared in B and referenced by some A_i are exported from B to A, for consistency reasons. This implies that test cases for A are generated solely by analyzing P's own functionality, i.e. the structure of $\mathcal{SE}$ and the predicates in $A_1, \dots, A_n$. In other words, B influences A's test case generation only as a whole and by its place in $\mathcal{SE}$. This means that the TTF will generate, at least, test cases that are going to make P to call Q from different places and with different parameters. For example, if the DNF tactic [8] is applied to $InsertAndSort_1$ there will be test cases that are going to test insert with an element belonging to the list and with one that does not. That is, these test cases will test whether or not insert correctly implements $x? \in \operatorname{ran} list$ and if it calls sort when it should. In a sense, this is all that it is worth to be tested of insert given that the correctness of the sorting algorithm implemented by sort was already tested. Indeed, for example, if tactics SP [8] and UQ [10] are applied to $InsertSort_1$, then sort will be tested with empty and non-empty lists of several lengths and where $x?$ belongs and does not belong to them.

6 Subroutines as Stubs of Themselves

The distinction between the *uses* and *calls* relations reduces the need for manually crafted stubs (cf. last paragraph in Sect. 4). However, a stub of Q is still needed when $\mathsf{P}\,\overline{uses}\,\mathsf{Q}$. One way to avoid building a stub of Q would be to use Q itself, but it cannot be done because Q is not proven correct, it was just tested. Nevertheless, if Q has passed some tests then we can be sure that it is correct for those inputs. Now, if P is tested in such a way that Q is always called as when it was tested, then Q itself can be used as stub. Furthermore, the cause of an error found during P's testing can only be blamed to P since Q has been "tested correct" for those inputs. We have made an attempt to formalize these ideas, thus yielding the basis for the mechanization of the search of those subroutines that can be stubs of themselves.

We have proved a theorem that gives conditions for a subroutine to be used as stub. Its proof relies on the uniformity hypothesis as stated in [17, page 17]. In order to state Theorem 1 we need a little bit of notation. Consider schemas $A, A_1, \dots, A_n$ and B like in (†). According to Sect. 5 only $A_1, \dots, A_n$ are unfolded in A. Let $vars(A)$ be the set of the variables declared in schema A^{VIS} that are declared in at least one A_i. That is, $vars(A)$ does not include variables declared only in B. If a is a test case derived from schema A and B is another schema, then $B^A(a)$ means the substitution of variables in $vars(B) \cap vars(A)$ by the values of the same variables in a (recall, from Sect. 3, that a test case in the TTF is a conjunction of equalities between variables in the *VIS* and constant values). We will note $A^A(a)$ simply as $A(a)$.

Theorem 1 assumes that A performs only one state change. This is the case, for instance, of $\mathit{InsertAndSort}_1$ in Sect. 2. See [11] for a theorem dealing with two state changes (one for P and one for Q).

Theorem 1. *Let P and Q be two subroutines such that $\mathsf{P}\,uses\,\mathsf{Q}$ and let A and B be their Z specifications, which in turn comply with (†). Assume there is just one state change in A. Let $B_1, \dots, B_n$ be the leaves of the testing tree generated by applying the TTF to B. Assume Q has passed all the tests derived from all these test specifications. Let a be a test case for P derived from A. If there is a B_j such that $B_j^A(a) \neq \emptyset$, then Q can be used as a stub when P is tested on a.*

Proof. If there is one state change in A then Q executes with the same values than P for variables in $vars(A) \cap vars(B)$.

If $B_j^A(a) \neq \emptyset$, then there is $b \in B_j$ such that a and b are equal on variables in $vars(A) \cap vars(B)$. Since Q has passed all its tests then it has passed a test from B_j. By the uniformity hypothesis Q is also correct on $b \in B_j$. Therefore, when P is executed on a, Q will be executed on b, thereby returning a correct answer to P. So Q can be used as a stub when P is tested on a. $\square$

This theorem will be further discussed in Sect. 8.

7 Errors Detected During Integration Testing

Leung and White give a classification of errors that can be detected during integration testing [20, 21]. They try to make a distinction between those errors that could have been detected during unit testing and those that are specific to integration testing. Below we briefly explain each of these errors and show that the TTF extended to integration testing can detect them.

Interpretation errors. There are three subclasses of these errors.

- Wrong function errors (WFE). Q does not provide the functionality indicated by its specification and P does not know that.

  Given that the TTF (and other MBT methods) generates test cases for Q from its specification, then WFEs will be detected when Q is tested as a unit. In other words, if a test case for Q, generated by the TTF, finds an error in Q this is an indication that it does not provide the functionality indicated by its specification
- Extra function errors (EFE). Q provides more functionality than P needs. P's developers know this but they wrongly implement P making it to call these extra functions.

  The TTF will generate at least one test case for each of the functionalities in the specification of Q. For instance, testing tactics such as DNF and FT will be very useful [8]. If P is tested in such a way that Q is called as to exercise all these functionalities, then P's problems will surface (because the extra functions will be called). In other words, it is necessary to apply the TTF to A in such a way that it generates enough test cases for P which will make it call Q in such a way that executes all its functionalities. In turn, this will be achieved if test cases derived from A verify the following theorem (the proof is omitted for brevity).

  Theorem 2. Let $a_1, \dots, a_m$ be the test cases for P; and $B_1, \dots, B_n$ be the leaves of the testing tree of Q. Assume these leaves represent all the functionalities provided by Q. The TTF will detect all EFE in P if for each $j \in 1 \mathinner{..} n$ there exists $i \in 1 \mathinner{..} m$ such that $B_j^A(a_i) \neq \emptyset$.

- Missing function errors (MFE). Inputs used by P to call Q are outside the domain of Q making it to behave unexpectedly.

  If B is total then P cannot make Q behave unexpectedly because there is a specified behavior for each input expected by Q. If B is partial then P should call Q with $b \notin B^{VIS}$ to execute it outside its input domain. But from b the input for P, a, must be found. It is easier to calculate a if A performs only one state change. For this case, we define a new testing tactic, called MF, that should be applied to operations whose corresponding subroutines are in the domain of the *uses* relation. The test specifications generated by MF are: $A_1^{MF} \mathrel{\widehat=} [A^{VIS} \mid \exists x_1, \dots, x_n \bullet \mathrm{pre}\ B]$ and $A_2^{MF} \mathrel{\widehat=} [A^{VIS} \mid \exists x_1, \dots, x_n \bullet \neg\ \mathrm{pre}\ B]$, where $x_1, \dots, x_n$ are the variables declared in B but not in A. Note that MF is applied to A, not to B but B is part of A as in (†). The TTF then encourages to further partition these test specifications by applying more testing tactics. Certainly MF will help to discover MFE because it will force P to call Q outside its domain due to A_2^{MF}.

Miscoded Call errors. P calls Q from wrong places. There are three subclasses.

– Extra call instruction (ECI). The calling instruction is placed on a path that should not contain such invocation.
– Wrong call instruction placement (WCI). The call is located on the right path, but in a wrong place.
– Missing instruction (MIC). Missing call on a path that should contain it.

Detecting these errors is one of the reasons for defining A as in (†). If A specifies exactly all the calls that P should make to Q, then the TTF will help to discover all of these errors. In effect, DNF applied to $\mathcal{SE}(B, A_1, \dots, A_n)$ will generate test specifications for all the situations where Q is called and those where it is not; other tactics, such as FT and UQ, will generate more detailed conditions under which Q is called. For example, when DNF is applied to $\mathit{InsertAndSort}_1$ it will generate a test specification characterized by the precondition under which Sort_1 is called and another characterized by its negation. Then, if insert does not call sort in the first case (MIC) the result will be $\mathit{list}' = \mathit{list}$ when it should be $\#\mathit{list}' = \#\mathit{list} + 1$; if it calls sort in the second case (ECI), the result will be $\#\mathit{list}' = \#\mathit{list} + 1$, when it should be $\mathit{list}' = \mathit{list}$.

Global errors (GER). These are errors related to the wrong use of global variables [20]. If $\mathsf{P}\ \mathit{uses}\ \mathsf{Q}$ but $\mathsf{P}\ \not{\mathit{calls}}\ \mathsf{Q}$, it means that they interact through a shared resource that can be thought of as a global variable, g. In this case Q defines a value for g that is later used by P. If this value is not what P expects, then P may fail. There are two causes that can make P to find an unexpected value in g: (a) Q does not verify B; or (b) Q does verify B but P assumes Q implements a different specification, say $\widehat{B}$.

In analyzing how the TTF can detect GER we will assume that $\mathsf{P}\ \mathit{uses}\ \mathsf{Q}$ but $\mathsf{P}\ \not{\mathit{calls}}\ \mathsf{Q}$, because when also $\mathsf{P}\ \mathit{calls}\ \mathsf{Q}$, all the previous results apply. If (a) causes the error, then it reduces to WFE because it means to see whether Q verifies its specification. Therefore, the true problem of integration testing regarding global variables is given under the following conditions: $\mathsf{P}\ \mathit{uses}\ \mathsf{Q}$ but $\mathsf{P}\ \not{\mathit{calls}}\ \mathsf{Q}$ and Q verifies B but P assumes Q implements a different specification, $\widehat{B}$. One possible way of detecting these errors is by executing Q before P while testing P. This way, however, complicates P's testing because now it is necessary to run other units before it, and they must be run in such a way as to make P fail.

Hence, we propose a different approach based on specification verification rather than on testing. In effect, the problem is a mismatch at the specification level, causing errors at the implementation level. That is, A assumes $\widehat{B}$ rather than B, so the problem is to find out this wrong assumption. If the involved operations are proven to verify some properties (state invariants, for instance) then these wrong assumptions will be detected. In this way, B will be changed for $\widehat{B}$ and it will become Q's specification. Therefore, $\widehat{B}$ cannot be wrong with respect to A, because the proven properties act as a common consistency ground for them. Then, if Q verifies $\widehat{B}$ it cannot set a wrong value for g from P's perspective. From here, all reduces to ensure that P and Q implement their specifications which means performing a thorough unit testing of each of them in isolation from each other. This is why in Sect. 4 we proposed not to include B in A when P *uses* Q but P $\not{\mathit{calls}}$ Q.

8 Discussion

Although we are interested in extending the TTF to integration testing, our results use only some of its details. Therefore, they can be used in other specification languages and MBT methods. Most of the results are based on fundamental concepts of Software Engineering like the *uses* relation, first-order logic and MBT in general.

Describing operations as in (†) is not a severe restriction on the use of the language and it has a non negligible impact on the application of the TTF to integration testing. The form of (†) makes it possible to automatically calculate the *uses* relation in the relevant cases for integration testing. That is, all the ordered pairs belonging to $\overline{uses}$ can be automatically computed. In turn, organizing integration testing around the *uses* relation provides several places for optimizing this process. The first one is given by the definition of the family of sets $\mathfrak{U}_i$. If integration is based on these sets then many errors can be caught with as less units already integrated as possible. The definition of the families of sets $\mathfrak{U}_i^k(j)$ provides a finer level for guiding integration testing. All this aims at making the search for the cause of an error as simple as possible, discarding errors as earlier as possible.

The fact that *uses* would have an important impact on reducing the costs of testing and that it can be automatically computed from a Z specification, might turn Z and *uses* more cost-effective. In this way they will be used not only as essential documents but they will be reused during testing as well.

Testing a unit in isolation is a rather ambiguous statement. In effect, if P *uses* Q, what it means testing P in isolation? If it means not using Q but a stub of it, then unit testing is faced with the problem of building correct stubs. Manually-crafted stubs are not only error prone but costly [16, 4, 19]. The approach presented here also aims at reducing the costs of stub generation and at making them reliable enough as not introducing errors. If integration follows the *uses* relation and each unit is certified at least for the inputs used during its testing, then they can be used as stubs for themselves, provided they are always called as when they were tested. Furthermore, those stubs implied by the *calls* relation can be automatically built, as was discussed in Sect. 4. Theorem 1 gives rather simple conditions under which a subroutine can be used as a stub for itself—although they are probabilistic given that the proofs depend upon the uniformity hypothesis. In this way, we are trading the cost and risk of building stubs for the cost of describing the *uses* relation and applying Theorem 1, which is almost automatic in many cases—see below. Finally, if this theorem cannot be proved for a given test case of P, i.e. this test case satisfies no leaf used to test Q, it is an indication that Q was poorly tested because one of its callers will call it in a functional situation not covered during its testing.

The use of subroutines as stubs for themselves somewhat blurs the distinction between unit and integration testing. However, integration testing may find new errors that are difficult or impossible to find during unit testing, as was shown in Sect. 7. In fact, the TTF extended to integration testing can cope with almost all the errors classified by Leung and White. Z and the TTF enable a formal analysis of some of these classes of errors. Theorem 2 and testing tactic MF show that the TTF can be further extended to deal with particular issues of integration testing.

A case study applying all these results can be found in [11].

More Detailed Issues. In Theorem 1, proving that $B_j^A(x) \neq \emptyset$ involves either the evaluation of a constant Z predicate or solving a satisfiability problem. In effect, if $vars(B) \subseteq vars(A)$ then all the free variables in B_j will be replaced by constant values when $B_j^A(x)$ is calculated; otherwise, there will be free variables in $B_j^A(x)$. In the first case $B_j^A(x) \neq \emptyset$ can always be automatically solved; in the second case it is necessary to decided whether $B_j^A(x)$ is satisfiable or not. This problem is undecidable because $B_j^A(x)$ can be a first-order predicate over the set theory. However, Fastest uses advanced Constraint Logic Programming techniques (the $\{log\}$ tool) for solving these predicates with very good results for real specifications [**?**,8]. Then, even when $B_j^A(x)$ has free variables Theorem 1 can be automatically applied in many situations.

9 Related Work

There is a lot of research on integration testing, from a MBT perspective [1, 24, 5, 3, 14, 16] or not [13, 2, 6, 22, 19, 18, 21], but we could not find articles analyzing in detail how Parnas's design theory and the Z notation can be used for integration testing. Clements and others [7, pages 68–71] pay attention to the *uses* relation and remark its importance in integration testing. In particular they say it can be used to narrow the search for the cause of an error found during integration testing but they do not go any deeper.

Leung and White [20, 21] study integration testing in the context of regression testing. Although they use the *calls* relation, they define sets of test cases to test subroutines during integration testing that have some similarities to those presented here. Apparently they are not interested in the stub generation problem, but in reducing the number of tests during regression.

Benz [5] acknowledges the fact that critical relationships for integration testing are not explicitly modeled and that MBT methods applied to integration testing may yield large state spaces. In his work Benz uses task models for specifying the interaction between components. Ali et al. [3] use UML collaboration diagrams to model interactions among classes and Statecharts for specifying their behavior. They propose a list of mutation operators that can be used to assess the effectiveness of integration testing methods. Since this list is aimed at object-oriented programs we preferred the taxonomy of errors proposed by Leung and White, also used by Orso [22]. Class State Machines (CSTM) are used by Gallagher, Offutt and Cincotta as the specification method for classes of object-oriented programs. These CSM are then combined into a component flow graph which is used to derive integration tests.

Testing components that can only be accessed through a system interface is the goal of the work by Schätz and Pfaller [24]. They use transition systems to model the behavior of components and hierarchical transition systems to model component interactions. The authors define the notion of Satisfied Integrated Test Case which plays a similar role as Theorem 1 in the present work. Another work that focuses on a specific problem, carving and replay based integration testing, is that of Elbaum and his colleagues [13]. However, the four steps of unit testing they use are the same used in Fastest: identify a program state, set it, execute the unit from it and evaluate the results.

Hartmann, Imoberdorf and Meisinger [16] use a method based on category partition to generate test cases from UML Statecharts specifying the behavior of components whose interactions are described be means of concepts borrowed from CSP. Category partition is essentially what the TTF does with the *VIS* of a Z operation. The authors aim at the stub generation problem but is not clear to us how their method reduces the number of manually-crafted stubs.

Labiche et al. [19] define an integration strategy based on class diagrams with the goal of minimizing the stub generation problem. Essentially they test a class after the classes it depends on. Labiche's integration order is an extension of Kung's [18] when dynamic dependencies and abstract classes are present. However, class or similar diagrams seldom include the functional specification of classes. In fact, these methods make a syntactic analysis of these diagrams resulting in a larger number of dependencies because they include not only "used" classes but also "called" classes.

10 Conclusions and Future Work

The TTF has been extended to integration testing providing, in principle, a good coverage during this level of testing because it covers almost all the errors in Leung and White's classification. Organizing integration testing around the *uses* relation shows several advantages that should be further investigated. The favorable impact that *uses* has on testing may make developers to describe it thereby reusing a key design document. Moreover, if a logical specification is cleverly structured, *uses* can be computed automatically. The extension minimizes the need for manually-crafted stubs by giving simple conditions that say when a stub can be automatically generated or when a subroutine can be used as a stub of itself.

However, it should be investigated what testing tactics should be applied to two subroutines belonging to the *uses* relation to prove Theorem 1 for all test cases, while still providing good unit coverage for both of them. Another issue that should be studied is the relation of Z's θ operator and operation promotion with integration testing.

Acknowledgments

Ana Cavalcanti made a number of corrections and suggestions to an early version of this paper. We thank her a lot for that.

References

1. Aiguier, M., Boulanger, F., Kanso, B.: A formal abstract framework for modelling and testing complex software systems. Theor. Comput. Sci. 455, 66–97 (Oct 2012), `http://dx.doi.org/10.1016/j.tcs.2011.12.072`
2. Alexander, R.T., Offutt, A.J.: Criteria for testing polymorphic relationships. In: Proceedings of the 11th International Symposium on Software Reliability Engineering. ISSRE '00, IEEE Computer Society, Washington, DC, USA (2000), `http://dl.acm.org/citation.cfm?id=851024.856208`
3. Ali, S., Briand, L.C., Rehman, M.J.u., Asghar, H., Iqbal, M.Z.Z., Nadeem, A.: A state-based approach to integration testing based on uml models. Inf. Softw. Technol. 49(11-12), 1087–1106 (Nov 2007), `http://dx.doi.org/10.1016/j.infsof.2006.11.002`
4. Baresi, L., Pezzè, M.: An introduction to software testing. Electron. Notes Theor. Comput. Sci. 148(1), 89–111 (Feb 2006), `http://dx.doi.org/10.1016/j.entcs.2005.12.014`
5. Benz, S.: Combining test case generation for component and integration testing. In: Proceedings of the 3rd international workshop on Advances in model-based testing. pp. 23–33. A-MOST '07, ACM, New York, NY, USA (2007), `http://doi.acm.org/10.1145/1291535.1291538`
6. Buy, U., Orso, A., Pezze, M.: Automated testing of classes. In: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis. pp. 39–48. ISSTA '00, ACM, New York, NY, USA (2000), `http://doi.acm.org/10.1145/347324.348870`
7. Clements, P., Garlan, D., Bass, L., Stafford, J., Nord, R., Ivers, J., Little, R.: Documenting Software Architectures: Views and Beyond. Pearson Education (2002)
8. Cristiá, M., Albertengo, P., Frydman, C., Plüss, B., Monetti, P.R.: Tool support for the test template framework. Software Testing, Verification and Reliability 24(1), 3–37 (2014), `http://dx.doi.org/10.1002/stvr.1477`
9. 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. pp. 268–277. IEEE Computer Society (2010)
10. Cristiá, M., Frydman, C.S.: Extending the Test Template Framework to deal with axiomatic descriptions, quantifiers and set comprehensions. In: Derrick, J., Fitzgerald, J.A., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., Riccobene, E. (eds.) ABZ. Lecture Notes in Computer Science, vol. 7316, pp. 280–293. Springer (2012)

11. Cristiá, M., Mesuro, J., Frydman, C.: Extending the Test Template Framework to integration testing, `https://www.dropbox.com/s/8dlyu2mctmzw57m/ttf-integration-testing.pdf`
12. Cristiá, M., Rodríguez Monetti, P.: Implementing and applying the Stocks-Carrington framework for model-based testing. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM. Lecture Notes in Computer Science, vol. 5885, pp. 167–185. Springer (2009)
13. Elbaum, S., Chin, H.N., Dwyer, M.B., Dokulil, J.: Carving differential unit test cases from system test cases. In: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering. pp. 253–264. SIGSOFT '06/FSE-14, ACM, New York, NY, USA (2006), `http://doi.acm.org/10.1145/1181775.1181806`
14. Gallagher, L., Offutt, J., Cincotta, A.: Integration testing of object-oriented components using finite state machines: Research articles. Softw. Test. Verif. Reliab. 16(4), 215–266 (Dec 2006), `http://dx.doi.org/10.1002/stvr.v16:4`
15. Ghezzi, C., Jazayeri, M., Mandrioli, D.: Fundamentals of software engineering (2nd ed.). Prentice Hall (2003)
16. Hartmann, J., Imoberdorf, C., Meisinger, M.: UML-based integration testing. In: Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 60–70. ISSTA '00, ACM, New York, NY, USA (2000), `http://doi.acm.org/10.1145/347324.348872`
17. Hierons, R.M., Bogdanov, K., Bowen, J.P., Cleaveland, R., Derrick, J., Dick, J., Gheorghe, M., Harman, M., Kapoor, K., Krause, P., Lüttgen, G., Simons, A.J.H., Vilkomir, S., Woodward, M.R., Zedan, H.: Using formal specifications to support testing. ACM Comput. Surv. 41(2), 1–76 (2009)
18. Kung, D.C., Gao, J., Hsia, P., Lin, J., Toyoshima, Y.: Class firewall, test order, and regression testing of object-oriented programs. JOOP 8(2), 51–65 (1995)
19. Labiche, Y., Thévenod-Fosse, P., Waeselynck, H., Durand, M.H.: Testing levels for object-oriented software. In: Proceedings of the 22nd International Conference on Software Engineering. pp. 136–145. ICSE '00, ACM, New York, NY, USA (2000), `http://doi.acm.org/10.1145/337180.337197`
20. Leung, H.K.N., White, L.: Insights into testing and regression testing global variables. Journal of Software Maintenance 2(4), 209–222 (1990)
21. Leung, H.K.N., White, L.: A study of integration testing and software regression at the integration level. In: Conference on Software Maintenance-90. pp. 290–301. San Diego, CA (1990)
22. Orso, A.: Integration Testing of Object-Oriented Software. Ph.D. thesis, Politecnico di Milano, Milan, Italy (february 1999)
23. Parnas, D.L.: Designing software for ease of extension and contraction. In: ICSE '78: Proceedings of the 3rd international conference on Software engineering. pp. 264–277. IEEE Press, Piscataway, NJ, USA (1978)
24. Schätz, B., Pfaller, C.: Integrating component tests to system tests. Electron. Notes Theor. Comput. Sci. 260, 225–241 (Jan 2010), `http://dx.doi.org/10.1016/j.entcs.2009.12.040`
25. Sommerville, I.: Software Engineering. Addison-Wesley, Harlow, England, 9th edn. (2010)
26. Stocks, P., Carrington, D.: A Framework for Specification-Based Testing. IEEE Transactions on Software Engineering 22(11), 777–793 (Nov 1996)