B Model Animation for External Verification

Hélène Waeselynck
LAAS-CNRS
7, Av. du Colonel Roche
31077 Toulouse Cedex 4, FRANCE
Email: waeselyn@laas.fr

Salimeh Behnia
LAAS-CNRS and INRETS
7, Av. du Colonel Roche
31077 Toulouse Cedex 4, FRANCE
Email: behnia@laas.fr

Abstract

The B method is a model-based approach covering all the software development process, from the specification to the code. External verification of B models aims to determine whether they correctly capture the informal requirements. It is argued that verification techniques like B model animation or code testing should accompany the formal development process and give a feedback of the system that is actually being specified. A uniform testing framework, irrespective of whether the input cases are executed on the final code or on the formal models, is presented. A B development process is considered as a series of stages where concrete models are built gradually based on the more abstract ones, the final code being just a compiled version of the most concrete model. A definition of test correctness, related to the one of refinement, is introduced. The consequences in terms of required animation facilities are discussed.

1 Introduction

The B method [1] is a formal approach covering all the software development process, from the specification to the code, through a series of proved refinement steps. There are now several examples of industrial applications of the approach: B is currently used in France for the development of safety-critical software for railway systems [3].

Previous work [14] has discussed the role of testing in the B formal development process. A first problem is that of the *validation of validation* [12], or how to reach confidence in the methods and tools used in building confidence in the system. A second problem concerns the *handling of the proof obligations that are not discharged* by the automatic prover: when the proofs are too complex, alternative verification techniques, like testing, may be used. The third problem is that of the *external verification* [10] of the B models, in order to determine whether the user needs are correctly addressed by the formal specification of the service to be delivered.

This paper is focused on the last problem. It is argued that external verification is needed not only during integration of software with the target hardware and equipment, but also during the formal development process. In this latter case, the aim is to track down specification faults originating from a misunderstanding of the functional requirements, or from the failure to express adequately an understood requirement. In essence, there is no way to prove that the B models correctly capture the informal requirements. Hence, other verification techniques, like *B model animation* or *code testing*, must be used. B model animation should allow early validation on an abstract level: as pointed out by [8], executable specifications are a useful means to give feedback to users on the behavior of the future system. However, an executable interpretation of the notation must be available.

The work presented in this paper aims to provide a uniform testing framework, irrespective of whether the input cases are executed on the final code or on the formal models. We think it is most fruitful to consider the B development process as a series of stages where more and more concrete models of the application are built, the final code being just a compiled version of the set of B implementations. This leads us to investigate a definition of *test correctness* that can be related to the one of refinement. If, at some stage of the development, the animation of B models supplies correct results with respect to the user needs, then the proof obligations prescribed by the B method should ensure that subsequent refinements also provide correct results in response to the same input cases. This imposes constraints on the way the B models have to be made executable: we discuss the facilities that should be offered by animation tools in order to support the uniform framework.

A brief overview of the B method is given in Section 2. The problem of external verification is exemplified by a small case study in Section 3. In Section 4, the notion of test correctness is stated in relation to the one of refinement, and the consequences in terms of required animation facilities are discussed in Section 5.

2 Overview of the B Method

The B method due to J-R. Abrial [1] is a model-based approach for the incremental development of specifications and their refinements down to an implementation. Proof obligations accompany the construction of the software.

The *abstract machine* is the basic element of a B development. It characterizes a machine which has an invisible memory and a number of keys. The values stored in the memory form the *state* of the machine, whereas the various keys are the *operations* that a user is able to activate in order to modify the state. A unique Abstract Machine Notation (AMN) is used for the description of machines at various levels of abstraction, in MACHINE, REFINEMENT and IMPLEMENTATION components.

The *declarative part* of a component describes its encapsulated state according to set-theoretic model and first order logic. The invariant states the static laws that the data must obey, whatever the operation applied.

Various *composition clauses* are defined in the B method, in order to be able to develop large software systems. Refinement is introduced by means of the REFINES clause. As soon as the refined version of an abstract machine becomes too complicated, it is decomposed into smaller components, through the IMPORTS clause. The IMPORTS clause proceeds according to the layered paradigm: the importing machine uses the service offered by the imported ones. It is said that the operations of the importing machine are *implemented* on the lower layer machines. Those lower layer machines are then independently refined and decomposed into smaller components. Other clauses introduced in [1] are INCLUDES, USES, SEES and EXTENDS that enrich the formal text of a component according to specific composition rules.

The *execution part* of a component, which specifies the dynamics, contains the initialization and some operations which are described under the form of a pre-condition and an action. The corresponding syntactic structures are interpreted in the *generalized substitution language*. The generalized substitutions (see Fig. 1) are predicate transformers: [S]R denotes the weakest precondition for substitution S establish postcondition R. For example, for the simple substitution $x := x+1$ and the postcondition $x = 5$ to be established, we have: $[x := x+1]\ x = 5 \Leftrightarrow x+1 = 5 \Leftrightarrow x = 4$. To facilitate the development of abstract machines, syntactic sugar is introduced. For example, P | S is rewritten as PRE P THEN S END. The construct $x :\in E$ (pronounced "x becomes a member of E") is the non-deterministic choice $@z.(z \in E \Rightarrow x := z)$.

Preconditioned substitutions are related to the notion of *termination*. Given a substitution S, trm(S) denotes the predicate that holds if and only if S terminates:

$$trm(S) \Leftrightarrow [S](x = x)$$

Simple substitution	$x := E$	$[x:=E]\ R \Leftrightarrow$ replacing all free occurrences of x in R by E
Empty substitution or no-op	skip	$[skip]\ R \Leftrightarrow R$
Preconditioning	P | S	$[P \mid S]\ R \Leftrightarrow P \wedge [S]\ R$
Bounded choice	S [] T	$[S [] T]\ R \Leftrightarrow [S]\ R \wedge [T]\ R$
Guarded substitution	$P \Rightarrow S$	$[P \Rightarrow S]\ R \Leftrightarrow P \Rightarrow [S]\ R$
Unbounded choice	@ x . S	$[@ x . S]\ R \Leftrightarrow \forall x.[S]\ R$ where x is not free in R

x denotes a variable, E is a set theoretical expression, P is a predicate, S and T are generalized substitutions

Figure 1. A subset of generalized substitution

The concept of guarded substitutions is related to the one of feasibility. It would be possible to specify *non-feasible* substitutions able to establish any postcondition: this clearly happens when the guard cannot hold. Given a substitution S working with variable x, mir(S) (miracle) and its negation fis(S) (feasible) are defined as:

$$mir(S) \Leftrightarrow [S](x \neq x) \qquad fis(S) \Leftrightarrow \neg\ [S](x \neq x)$$

A miraculous substitution refines any substitution working with the same variables, but there is no feasible substitution refining it.

Checking the mathematical consistency of a MACHINE component involves proving that its initialization establishes the invariant and that each operation, called within its precondition, terminates and preserves the invariant. Checking the correctness of a REFINEMENT or an IMPLEMENTATION involves checking that the initialization and the operations preserve the semantics of their corresponding more abstract versions. The composition clauses allow the proof to be modular, abstract machines being constructed in an incremental fashion from smaller, already proved components.

3 External Verification: an Introductive Example

The Triangle case study is well-known to the testing community. Examples of its specification can be found in [6] (in Z notation) and [7] (in VDM notation). The required program is to input three natural numbers and determine whether these values can be the sides of a triangle, and if so, what type of triangle (equilateral, isosceles, rectangle or scalene). Let us recall that natural numbers x, y and z can be sides of a triangle if and only if they verify the triangle inequalities :

$$x < y + z \ \wedge\ y < z + x \ \wedge\ z < x + y$$

Although this example is clearly not representative of industrial practice, it is sufficient to illustrate some concepts of the B method. Also, we deliberately show the B development of the Triangle performed by a student: it provides us with a striking example of the possible

introduction of faults during a formal development. Then, we will draw some general conclusions concerning the problem of external verification for more realistic B projects.

Figure 2 shows the architecture of the Triangle B development. It is a layered architecture typical of B projects: a MACHINE is refined down to an IMPLEMENTATION; the IMPLEMENTATION uses the service of lower layer MACHINEs through the IMPORTS decomposition mechanism; each imported MACHINE is in turn refined and so on. We provide below a detailed description of the involved components, trying to identify which subset of B development should be the focus of external verification. All components being stateless, we show only their operations (see Fig. 3).

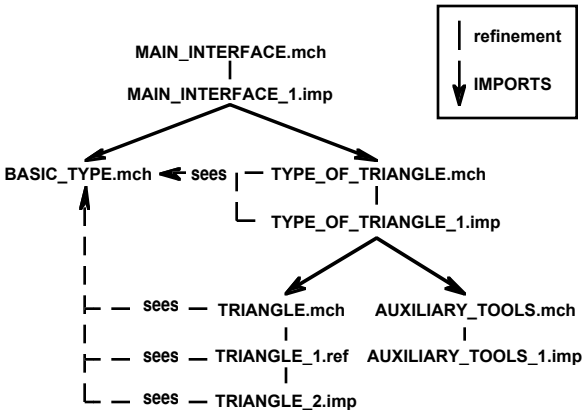


Figure 2. Architecture of the development of the Triangle case study

The root component is the MACHINE *MAIN_INTERFACE*. It provides a *main* operation which asks the user to enter three numbers representing sides of the triangle, launches the computation of the triangle type, and then outputs the result. The computation of the triangle type is performed through a call to one imported operation, *Classify_Triangle*, whose B source is given in Figure 3a. It can be seen that at this early development stage, all that is specified is the typing of input and output variables.

In the next stage, the proposed refinement of operation *Classify_Triangle*, shown in Figure 3b, consists of first deciding whether the input values constitute a triangle (operation *Is_a_Triangle*) and then, if the answer is positive, of determining the type of the triangle (operation *Which_Triangle*). In order to facilitate the task of *Is_a_Triangle* the numbers are first sorted (operation *Sort_3_Numbers*). If the operation *Is_a_Triangle* returns FALSE, the associated type will be INVALID.

The previous operations are separated in two categories: operations concerning a triangle (*Is_a_Triangle*, *Which_Triangle*) and auxiliary operations (*Sort_3_Numbers*), which are imported from MACHINE components *TRIANGLE* and *AUXILIARY_TOOLS* respectively. This approach corresponds to the principle of decomposing a problem into subproblems to be independently refined. The MACHINE *BASIC_TYPE* contains global definitions, like the set representing possible types of a triangle and operations concerning reading and writing types.

The first called operation is *Sort_3_Numbers*. Its specification in *TRIANGLE* (Fig. 3c) is loose. For example with xx = yy = 2 and zz = 1 as inputs, there are four possibilities for the outputs, including the intended one: (i) uu = vv = 1 and ww = 2; **(ii) uu = 1 and vv = ww = 2**; iii) uu = vv = ww = 1; iv) uu = vv = ww = 2.

Operation *Is_a_Triangle* (Fig. 3d) is intended to verify the triangle inequality: the sorted x, y and z are sides of a triangle if and only if x + y > z. This is not exactly what is stated in *TRIANGLE*. The output of the operation is loosely specified, due to the use of an "implies" connective in place of an equivalence one: actually, bb := FALSE would be a correct refinement of this operation. Some comments can also be made about the specification of *Which_Triangle* in Figure 3d. This operation is intended to associate a type to a valid triangle. Its precondition does not state that the triangle inequality must be previously verified and its body only contains typing information.

Having analyzed the called operations, it is now possible to conclude on the proposed refinement of *Classify_Triangle* in Figure 3b. At this development stage, the proof obligations ensure that:

- all imported operations are called within their precondition;
- when expanding the calls to operations of the imported MACHINEs, this version of *Classify_Triangle* preserves the semantics of its more abstract version, that is, it returns a value from the set TRIANGLE_TYPES when invoked with three natural inputs.

Due to the looseness of the called operations, we do not have a model of the intended behavior yet. It is even difficult to understand the dynamics of what has been specified so far. Actually, the version of *Classify_Triangle* in Figure 3b states that the output result must be INVALID for the input case (0, 0, 0), and may be any type of triangle for the other input configurations. Further proved refinements of this operation are not ensured to fulfill the user requirements. Hence, external verification has to be performed on subsequent stages of development.

```
tt ← Classify_Triangle (xx, yy, zz) ≙
    PRE
      xx ∈ NAT ∧ yy ∈ NAT ∧
      zz ∈ NAT
    THEN
      tt :∈ TRIANGLE_TYPES
    END
```

(a) in MACHINE TYPE_OF_TRIANGLE

```
tt ← Classify_Triangle (xx, yy, zz) ≙
    BEGIN
      VAR tx, ty, tz, bb IN
        tx, ty, tz ← Sort_3_Numbers (xx, yy, zz);
        bb ← Is_a_Triangle (tx, ty, tz);
        IF bb = TRUE THEN
          tt ← Which_Triangle (tx, ty, tz)
        ELSE
          tt ← Copy_TYPE_OF_TRIANGLE(INVALID)
        END
      END
    END
```

(b) in IMPLEMENTATION TYPE_OF_TRIANGLE_1

```
uu, vv, ww ← Sort_3_Numbers (xx, yy, zz) ≙
        PRE
          xx ∈ NAT ∧ yy ∈ NAT ∧ zz ∈ NAT
        THEN
          ANY ua, va, wa WHERE
            ua ∈ NAT ∧ va ∈ NAT ∧ wa ∈ NAT ∧
            ua ∈ {xx yy, zz} ∧ va ∈ {xx, yy, zz} ∧
            wa ∈ {xx, yy, zz} ∧ ua ≤ va ∧ va ≤ wa
          THEN
            uu, vv, ww := ua, va, wa
          END
      END
```

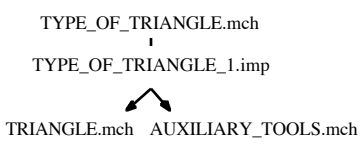
(c) in MACHINE AUXILIARY_TOOLS

```
bb ← Is_a_Triangle (xx, yy, zz) ≙
    PRE
      xx ∈ NAT ∧ yy ∈ NAT ∧
      zz ∈ NAT ∧
      xx ≤ yy ∧ yy ≤ zz
    THEN
      ANY b_t WHERE
        b_t ∈ BOOL ∧
        (b_t = TRUE => yy > zz - xx)
      THEN bb := b_t
      END
    END
```

```
ttype ← Which_Triangle (xx, yy, zz) ≙
        PRE
          xx ∈ NAT ∧ yy ∈ NAT ∧
          zz ∈ NAT ∧ xx ≤ yy ∧
          yy ≤ zz
        THEN
          ANY type WHERE
            type ∈ TRIANGLE_TYPES
          THEN
            ttype := type
          END
        END
```

(d) in MACHINE TRIANGLE

Figure 3. Operations in a subset of the Triangle project



The set of acceptable programs is now constrained both from a functional and a structural viewpoint: the algorithm of *Classify_Triangle* is the one given in Figure 3b, and further refinements can only consist in proposing more concrete versions of the operations called within this algorithm. *Which_Triangle* is the only called operation whose refinement down to an implementation is done in two steps. Its intermediate version in REFINEMENT *TRIANGLE_1* specifies the equilateral and isosceles case, but remains loose for the rectangle and scalene cases. Hence, for this B development of the triangle case study, there is no intermediate model of expected behavior before all operations have been refined down to an implementation. Faults may be introduced at *any* step of the formal development process without being exposed by refinement proof obligations. In that case, the adopted modeling approach does not allow external verification to be focused on a subset of the formal development: it must accompany the whole process.

As said at the beginning of this section, this case study is an extreme case for a B development. In the small examples given in [11], the specification and refinement phases are clearly separated. However, for realistic software systems, such a clear separation may not be practical for two main reasons:

- refinement steps may be necessary to introduce sequencing of operations, which cannot be expressed at the most abstract level;
- writing a (flat) complete specification and *then* starting the refinement and decomposition into layers often results in too complex proof obligations.

But there may be an intermediate stage, involving components in different levels of abstraction (MACHINE to IMPLEMENTATION), where all functional requirements have been captured. This is the modeling approach adopted by the French railway industry [3]: the informal specification is first captured during a preliminary B design phase using both refinement and decomposition into layers; the resulting model is a subtree of the final architecture of B components. Once this model has been validated by thorough review, the detailed B design phase consists in further expanding the leaf nodes of the subtree down to IMPLEMENTATIONs. So, in typical B developments, the smallest meaningful model with respect to the functional requirements is likely to involve several levels of refinement and to traverse several logical layers. This model should be the target of external verification.

4 Uniform Testing Framework

Testing methods are one of the possibilities to perform external verification [10]. Testing consists in exercising the target piece of software by supplying it with a sample of input values (see e.g. [4]). Checking whether the output results conform to the requirements is known as the test *oracle* problem [16]. Since our aim is testing for external verification of the B development process, it is worth noting that the expected output results cannot be determined from the B specifications.

The B development process can be seen as a series of stages where more and more concrete models of the application are built. As shown in the previous section, some user requirements may be captured only at the end of the development. Then external verification may be done by testing the final code, which can be seen as a compiled version of the most concrete model. But it is better methodology to express the requirements earlier in the development. This also permits external verification to be performed earlier, provided that there is a means to test abstract models as well. Several tools supporting B model animation already exist or are under development. Hence the idea of investigating a uniform testing framework, irrespective of whether the input cases are executed on the final code or on the formal models.

Our uniform framework covers two aspects: (i) identification of the development stages and (ii) formalization of the notions of test sequence and test oracle for the corresponding models. The problem of test data selection is not addressed in this paper. We do not consider whether to derive the test input sets from the functional analysis of user requirements, or from the structural analysis of the B models. Rather, we concentrate on the following issue: for a given test set, does acceptance of the results supplied by a model imply acceptance of the results supplied by proved refinements of this model?

4.1 Identification of Development Stages

The B method calls for the incremental development of abstract machines. The formal text corresponding to an abstract machine is split in smaller components linked by composition clauses (REFINES, SEES, IMPORTS, …): in order to be able to reason about the observable behavior specified in this way, we need to flatten the involved components and obtain the equivalent abstract machine. Such a model flattener is not yet provided by the existing B tool sets. A recursive algorithm has been proposed in [15]: it takes as input a data structure representing a tree of components, and produces the corresponding flattened formal text. The valid input domain of the model flattener is derived from the analysis of the structuring mechanisms of the B method, allowing us to identify the conditions for a set of components to jointly form an abstract machine.

The identification of the development stages is based on this notion of flattening. Using the formal definition of refinement for abstract machines, it is possible to define a

partial order relation between flattened models (= abstract machines) obtained using our algorithm. An example using the Triangle case study is shown in Figure 4. It is worth noting that the refinement relation is more than a tree inclusion relation, because some additional conditions must be fulfilled by the set of B components being compared.

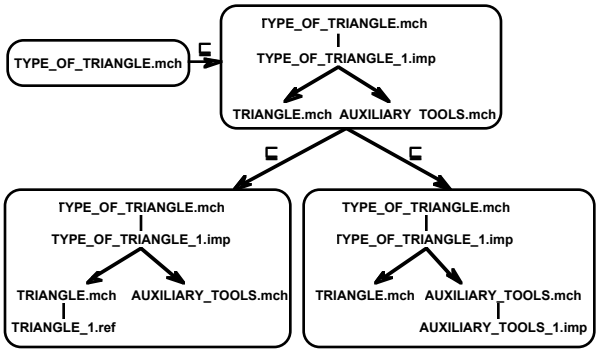


Figure 4. Partial order of flattened models from a subset of the Triangle development

Then we may identify a hierarchy of flattened machines that model the whole application, or a subsystem of it, at decreasing levels of abstraction: at the top of the hierarchy there is the model involving no REFINES or IMPORTS clauses; at the bottom of the hierarchy there is the final model to be automatically translated into a programming language; intermediate flattened models form the various development stages. All these flattened models define an observable behavior that may be tested for the purpose of external verification. However, whether the aim is to validate the complete preliminary design (i.e. the stage where all functional requirements are supposed to be captured), or a subset of it, the target model should be comprehensive enough to generate behavior patterns that are meaningful from the perspective of an end user.

4.2 Observable Behavior of an Abstract Machine

Our definition of test sequence and test oracle for flattened models are related to the notion of observable behavior of an abstract machine, defined in the B-Book in terms of services offered to "external" substitutions capable of being "implemented" on this machine. Let us recall that an abstract machine offers:

- an initialization allowing the encapsulated data to be put in a state satisfying the invariant;
- a set of operations preserving the invariant and proved to terminate within their most external precondition: op ≙ PRE P THEN S END is such that trm (S), whatever the additional preconditions involved in S.

The "external" substitution contains a finite sequence of calls to the operations of the machine, beginning with its initialization so that the machine is first put in a proper state satisfying the invariant. It contains no reference to the state variables encapsulated in the machine. As an example, let M be an abstract machine encapsulating a single state variable $v_M \in S_M$, and offering two operations: op1 (x) and z ← op2 (y). Then T is an example of "external" substitution capable of being "implemented" on M:

```
T ≙ BEGIN
        Init;
        op1 (value_x1);
        output_z ← op2 (value_y1);
        op1 (value_x2)
    END
```

Let T_M be the implementation of T on M: calls to Init, op1 and op2 are expanded by their respective definition in M, replacing the formal input and output parameters by the actual ones. Due to the hiding principle, the observable behavior of T_M is actually the one of:

$$@v_M \,.\, (v_M \in S_M \Rightarrow T_M)$$

The '@' quantifier (see Fig. 1) hides the effect of the substitution on the encapsulated variable v_M: all that can be observed concerns the establishment of postconditions on output_z. Suppose now that T is implemented on another abstract machine N having the same signature as M (same operation names, same input and output parameters), but encapsulating a different state variable $v_N \in S_N$, and defining different initialization and operation bodies. Then, T will not be able to distinguish between M and N provided that:

$$@v_M \,.\, (v_M \in S_M \Rightarrow T_M) = @v_N \,.\, (v_N \in S_N \Rightarrow T_N)$$

that is, both substitutions establish the same postconditions on the observable parameter output_z. More generally, two machines having the same signature are observationally equivalent if and only if they cannot be distinguished by any external substitution capable of being implemented on them.

In a similar way, the refinement relation between two machines is introduced as follows in the B-Book:

$$@v_M \,.\, (v_M \in S_M \Rightarrow T_M) \sqsubseteq @v_N \,.\, (v_N \in S_N \Rightarrow T_N)$$

for *each* external substitution T executed on M and N in the form of T_M and T_N respectively, where $\sqsubseteq$ denotes the classical algorithmic refinement relation (weakening of the precondition, and diminishing of the non-determinism).

This definition is not very practical from the perspective of proving a B development, because it involves an informal quantification over each external substitution. It is demonstrated in the B-Book that it can be replaced by proof obligations referring only to M and N: this is done at the expense of forcing the specifier to find a proper total relation linking the concrete states to the abstract ones. We refer here to the initial definition of refinement because it is most adequate from the perspective of testing.

4.3 Test Sequence and Test Oracle

As also noticed by [2] a *test sequence* can be seen as an external substitution T implemented on the target abstract machine M. It consists of a sequence of operation calls, the input parameters being instantiated with actual scalar *values*, and each output parameter being actualized by distinct scalar *variables*. We assume that the expected result of the test sequence, to be determined from user requirements, can be formalized as another substitution T_0 working with the same output variables: the semantics of T_0 determines the postconditions to be established for the output values "computed" using the operations of M. If the requirements are not loose, T_0 may be a sequence of simple substitutions giving a value to each output variable; if alternative results can be accepted, T_0 may involve non-deterministic choices in a set of values.

The main idea is to have the notion of test correctness be defined in terms of a refinement relation. Then the *test oracle* that compares expected and supplied results corresponds to a proof obligation that T_0 is refined by the implementation of the test sequence on M:

$$T_0 \sqsubseteq @v_M \,.\, (v_M \in S_M \Rightarrow T_M)$$

Due to the transitivity of the $\sqsubseteq$ relation, if an abstract machine supplies accepted results when exercised with T, then any proved refinement is bound to supply accepted results as well. Hence, as far as sequence T is concerned, there is a development stage where further testing of formal models would not bring any new information. As was exemplified by the Triangle case study, this development stage may be located anywhere between the topmost machine and the final model involving all implementations, depending on the adopted modeling approach. Note that conformance to expected results is defined as a refinement relation – not an equivalence one – because the tested model is allowed to be more precise than the requirements, in case the latter are loose and leave room for decisions.

Going back to the Triangle example (a very simple one since all components are stateless), one test case could be: $T \mathrel{\widehat{=}} res \leftarrow Classify\_Triangle\ (1, 1, 1)$
with $T_0 \mathrel{\widehat{=}} res := equilateral$.

It can be shown that MACHINE *TYPE_OF_TRIANGLE* does not fulfill this test requirement, since from Figure 3a:

$T_{TYPE\_OF\_TRIANGLE} \mathrel{\widehat{=}} res :\in TRIANGLE\_TYPES$

is obviously not a refinement of T_0. In this case, we are faced to the problem of under-specification ($T_{TYPE\_OF\_TRIANGLE}$ is strictly refined by T_0): correct refinements of *TYPE_OF_TRIANGLE* may fulfill the test requirement, but are not bound to. The other case leading the oracle to reject the results supplied by an abstract machine M is when

$$\neg\,(T_0 \sqsubseteq @v_M \,.\, (v_M \in S_M \Rightarrow T_M)) \wedge \neg\,(@v_M \,.\, (v_M \in S_M \Rightarrow T_M) \sqsubseteq T_0)$$

meaning that T has exposed an inconsistency between the formal model and the requirements.

5 Consequences of Uniform Framework

We have just seen that our uniform framework defines the testing of flattened abstract machines as follows: a test sequence is an external substitution to be implemented on the machine; the test oracle amounts to checking whether the implemented test sequence refines another substitution that assigns their expected value (or set of acceptable values) to the outputs. We will now discuss some practical consequences of these definitions.

5.1 Testing Via Proof Obligations

The definition of test oracle given in Section 4.3 implies that the testing of formal models could be entirely performed through B proof obligations. Provided that we have a model flattener that builds the abstract machine resulting from the target set of B components, we could proceed as in Figure 5 to implement each test sequence T returning observable outputs $o_1, \ldots, o_n$: specify T_0 and T as operations of two B components linked by a refinement relation. In component *Test_Driver_1*, the IMPORTS clause makes T be implemented on the target abstract machine, so that the refinement proof obligation of *Test_Driver_1* would provide us with the defined oracle.

However such a "testing" approach may be extremely costly. Let us recall that external verification is likely to be performed after several steps of refinement and decomposition into layers: the flattened model is expected to be complex and difficult to prove.

MACHINE Test_Driver OPERATIONS $o_1,\ldots,o_n \leftarrow$ test_sequence $\mathrel{\widehat{=}} T_0$ END	IMPLEMENTATION Test_Driver_1 REFINES Test_Driver IMPORTS Target_Abstract_Machine OPERATIONS $o_1,\ldots,o_n \leftarrow$ test_sequence $\mathrel{\widehat{=}}$ T END

Figure 5. Test experiment by means of B proof obligations

5.2 Testing Via Animation

The classical way of testing consists in getting output results from code execution, and in verifying that each supplied value is an acceptable one. In that case, the oracle may be a program performing simple checks like equality between observed and expected output values, or

membership of an observed value in a set of acceptable ones. If external verification is to be performed at an intermediate development stage, the output results have to be computed through model animation.

Since first-order logic and set theory are not expected to be completely executable, the animation tool may fail to generate output results in some cases, and exhibit non-terminating behavior. Then the test experiment is inconclusive for the target model. When animation terminates and delivers output results, the uniform framework relies on the assumption that these results are consistent with the ones that would be expected from the predicate transformer semantics: this is the problem of providing correct executable interpretation of the B notation. To clarify this notion, let us consider three aspects of generalized substitutions, namely non-determinism (looseness), termination, and feasibility.

Non-determinism. The interpretation of loose substitutions should give the set of *all* possible output values. Let us take the example of MACHINE *TRIANGLE* (Fig. 3d) supplied with the following test case:

$$T \cong oo \leftarrow Is\_a\_Triangle\ (1, 2, 2)$$

with the test requirement $T_0 \cong oo := TRUE$.

The predicate transformer semantics tells us that both TRUE and FALSE are possible values for oo. Hence $T_{TRIANGLE}$ is not a refinement of T_0. If the animation tool handles non-determinism by computing a single value (say, TRUE), this partial result could be wrongly accepted by the test oracle. The animation results for non-deterministic substitutions must always fall in one of the three following categories:

- The animation tool is unable to produce any results. Then the test experiment is inconclusive.
- The tool computes a subset of output values, and the test oracle is reported that the result is partial. Then the test experiment is inconclusive if the subset of output values is included in the set of acceptable ones.
- The tool computes the complete set of output values. Then the oracle is able to check that *each* possible value is a member of the set of expected ones.

Termination. If the test sequence is derived from some functional analysis independent of the B development, it may be the case that an operation is called outside its most external precondition: this would reveal a fault since the operations of the B model must terminate at least in the cases required by the test sequence. The animation tool must take this situation into account, and fail to produce any output value. Note that the action of a non-terminating substitution is allowed to be badly formed. For example, let us consider an operation involving a partial function f:

$$op \cong PRE\ x \in dom(f)\ THEN\ y := f(x)\ END.$$

If the operation is called outside its precondition, the actual value of x does not belong to the domain of the function. Then, the action part y := f(x) of the operation is meaningless, and no attempt should be made to interpret it.

Feasibility. There are no specific B proof obligations (PO) relating to feasibility because such existence proofs are preferably replaced by constructive ones. It would be possible for a MACHINE or REFINEMENT component to contain miraculous substitutions; but if the project ends up with a set of proved IMPLEMENTATIONs, then it is sure that such was not the case. Let T be a test sequence implemented on an abstract machine M. If $mir(T_M)$ holds, T_M is a correct refinement of any T_0 working with the same variables. Hence, if "testing" of formal models is performed through B refinement POs, as suggested in Section 5.1, then machine M would be accepted as correct. However, by definition, miraculous substitutions are not feasible: there is no correct executable interpretation of T_M producing an output value. The animation tool is expected either to exhibit non-terminating behavior, or to report that no output value is reachable: none of these cases would lead the oracle to conclude that the test results conform to the requirements. In that sense, the animation output checks departs from a refinement PO since $fis(T_M)$ is required.

The problem of providing correct executable interpretation of model-based specifications has been investigated in [5] for the Z notation. The authors propose an approach to establish the correctness of a Z animation tool. They make precise how computed results may be a proper approximation of the results that would be expected from the intended Zermelo-Fraenkel semantics. Their definition of correctness takes into account the fact that the tool may exhibit non-terminating behavior. The problems with loose specifications identified above are also addressed by their definition. It should be possible to transfer this work to the animation of B models.

Let us now discuss whether the commercialized B animators possess the desired properties. They are not based on the principle of executable interpretation: predicates and set theoretic expressions are evaluated using rewriting rules, often with the aid of the user, and the final effect of an operation may be displayed in a symbolic form. The non-determinism must always be resolved by the user, who is asked to enter one value. In our opinion, such tools are useful in order to have the specifier - not the validator - gain better insight into what is stated in the model, through exploration and what-if analysis. For example, with the Atelier B animator, it is possible to display the value of any variable but also of any predicate or expression entered by the user; connection to the prover is supported, so that termination of operations or preservation of the invariant can be proved on-line in the current context of animation. This should facilitate the diagnosis of fault(s) that prevent the POs of a component from being discharged. However, such tools are not

sufficient to support our testing framework. First, they do not handle intermediate models including refinement levels and logical layers: they work only for MACHINE components. More importantly, correctness is not ensured. The treatment of non-determinism does not fulfill the requirements for correctness. And since the user interferes on-line with the simplification of predicates and expressions, there is no guarantee that the semantics of substitutions is preserved.

We are aware of on-going academic work using constraint logic programming for B prototyping purposes [9]. It is hoped that such work will offer a suitable solution, at least for a subset of the B notation.

5.3 Limitations of Test Correctness Based on Refinement

Our testing framework defines the test oracle in terms of a refinement relation. The obvious purpose is to take advantage of the formality of the B method: given a test sequence, acceptance of the results for a model got at some stage of the development should imply acceptance of the results for subsequent proved refinements of this model. However, the B notion of correctness exhibits some limitations: we discuss below the problem of software systems having to interact with an external environment.

B models may interact with an external environment through specific interfaces. The interfaces are provided in libraries of MACHINE components which are not intended to be refined, but for which the corresponding code in programming languages is already available. No matter how sophisticated at the code level, all interface operations have similar abstract specifications in B notation:

- read operations are specified as a non-deterministic choice into a set of possible values; for example, ANY value WHERE value ∈ INT THEN input := value END specifies the acquisition of an integer data irrespective of the input device from which this data originates.
- write operations are specified as substitutions whose only observable effect is to terminate when the written parameter has the appropriate type; a typical example is: PRE to_be_sent ∈ SET_OF_COMMANDS THEN skip END.

Interactive inputs and outputs must be taken into account in the testing framework. When the final code is executed, it is obvious that interactive inputs can be made controllable and that interactive outputs can be made observable. When formal models are being animated, it is desirable to have the same situation.

Now, let us assume that the B animation environment offers a set of basic facilities to trace communication events, so that the test oracle can refer to them: it is no more true that acceptance of the results for one model implies the acceptance of the results for proved refinements of this model. The proof obligations of the B method do not cover interactive aspects, as shown below:

```
incr ≙ BEGIN
          VAR xx IN
             xx ← INTERVAL_READ (0, 9);
             INT_WRITE (xx + 1)
          END
       END
```

The substitution obtained by expanding the read and write operation calls by the corresponding formal text is equivalent to skip: *incr* always terminates and a weak postcondition TRUE is established (assuming a stateless component). Correct refinements of *incr* could be:

```
                                  VAR xx IN
STRING_WRITE ("Hello")     or       xx ← INTERVAL_READ (0, 5);
                                    INT_WRITE (xx + 2)
                                  END
```

among many other possibilities.

The definition of observable behavior recalled in Section 4.2 is too restrictive for abstract machines invoking interface operations: it may be desirable to view them as communicating systems, which calls for proper notation and semantics. Typically, a communicating system, and the test scenarios associated to it, are described as processes connected to each other. The underlying theory is outside the scope of the B method.

This limitation arises for systems exhibiting complex interaction schemes with their external environment. However, like the B method, the proposed framework is adequate for systems like the Triangle example, in which we can clearly identify three separate phases of data acquisition, processing, and display of results. Animation concentrates on the phase of data processing and stubs are developed to simulate input and output phases.

6 Conclusion

There is no way to prove that a formal model satisfies its "intended" behavior, even when specifications seem to be simple and self-explanatory at the first sight. Hence, other verification techniques are still needed, like testing. Testing is a partial verification technique that involves sampling the input domain to verify the conformance of the software product to the user's needs. As recalled by [10], testing and proving are probably complementary, just as proofs and refutations are complementary.

The B development process can be seen as a series of stages where more and more concrete models of the application are built, the final code being at the bottom of the hierarchy. Hence, it is suggested that external verification should begin as soon as possible, through model animation. For animation to be useful, it is desirable to take into account the B proof obligations in the underlying framework. Given a test sequence,

acceptance of the results supplied by a model should imply acceptance of the results supplied by proved refinements of this model. Moreover, since the aim is external verification, the animated model should be comprehensive enough to generate behavior patterns that are meaningful from the perspective of an end user.

The notions of test sequences and expected outputs have been expressed within the framework of the B method. This allows us to define the test oracle in terms of a refinement relation. Exploring the consequences of this framework, we found that current B animation tools are not yet satisfactory for our purpose, especially regarding the treatment of loose specifications. Research has been done as for providing correct executable semantics for Z specifications [5]. This work should be applicable for the animation of B models. Conversely, we believe that our work should be transferable to other model-based methodologies with refinement: for example, an expression of test cases as Z schemas has already been proposed in [6].

A limitation of the proposed framework is the treatment of interactive inputs and outputs. The granularity of observation is an operation. Interactive I/O taking place inside an operation are neither observable nor controllable. The refinement process does not allow us to establish any relation between the interactive behavior of an abstract machine and the one of its refined version. This is due to the fact that the semantics of interactive I/O is not covered by the B method, making it inadequate for the development of highly interactive software. Like the B method, the proposed framework is adequate if the application has three distinct phases of data acquisition, processing and output.

In typical B developments, the topmost machine component is not meaningful with respect to the functional requirements. The smallest meaningful model is likely to involve several levels of refinement and to traverse several logical layers, hence the need for a model flattener. In order to identify the conditions for a set of B components to jointly form an abstract machine, the structuring mechanisms of the B method have been analyzed in [15]. Flattened abstract machines can be partially ordered according to a refinement relation, which allows us to clarify the notion of development stages. Based on this, it will be possible to work on the definition of test integration strategies, or to compare the models obtained at the various stages of the development in order to analyze whether or not new functional features have been captured.

Our work continues by studying the problem of test sequence selection. The first step is to study the structural coverage of B models, being aware that this will not be the only considered strategy. Following our uniform framework, we will investigate structural criteria that can be applied to an abstract model [7] as well as a concrete one [13].

Acknowledgment. The very first idea of performing oracle checks through proof obligations was jointly introduced by Hélène Waeselynck and Jean-Louis Boulanger in 1995, while they were both working at INRETS. J-L. Boulanger was also the author of the Triangle B development, in his early beginning with the B method.

References

[1] J.R. Abrial, *The B-Book – Assigning programs to meanings*, Cambridge University Press, 1996.

[2] S. Alnet, "Test de programme à partir de spécifications B: les problèmes", Studentship Report, LRI - University of Paris Sud (France), September 1996.

[3] P. Behm, P. Desforges, F. Mejia, "Application de la méthode B dans l'industrie ferroviaire", in *Application des techniques formelles au logiciel*, ARAGO 20, OFTA, ISBN 2-906028-06-1, 1997, pp. 59-87.

[4] B. Beizer, *Software Testing Techniques,* 2nd Edition, Van Nostrand Reinhold, New York, 1990.

[5] P.T. Breuer, J.P. Bowen, "Towards correct executable semantics for Z", in *Proc. of the Z User Workshop*, 1994, pp. 185-209.

[6] D. Carrington, P. Stocks, "A tale of two paradigms: Formal methods and software testing", in *Proc. of the 8th Z User Meeting*, Cambridge (UK), June 1994, pp 51-68.

[7] J. Dick, A. Faivre, "Automating the generation and sequencing of test cases from model-based specifications", in *Proc. of the Int. Formal Methods Europe Symposium*, 1993, pp. 268-284.

[8] N.E. Fuchs, "Specifications are (preferably) executable", *Software Engineering Journal*, Vol. 7, n°5, September 1992, pp. 323-334.

[9] M. Fumey, "Animation de spécifications formelles B en programmation logique avec contraintes ensemblistes", Studentship Report, LIB - University of Franche-Comté (France), September 1997.

[10] M.C. Gaudel, "Advantages and limits of formal approaches for ultra-high dependability", in *Proc. of the Int. Workshop on Software Specification and Design*, October 1991.

[11] K. Lano, H. Haughton, *Specification in B: An introduction using the B Toolkit*, Imperial College Press, 1996.

[12] J.C. Laprie (Ed.), *Dependability: basic concepts and terminology*, Springer Verlag, Vienna (Austria), 1992.

[13] S.C. Ntafos, "A Comparison of Some Structural Testing Strategies", *IEEE Transactions on Software Engineering*, Vol. 14, n° 6, June 1988, pp. 868-874.

[14] H. Waeselynck, J.L. Boulanger, "The role of testing in the B formal development process", in *Proc. of the 6th Int. Symposium on Software Reliability Engineering*, 1995, pp. 58-67.

[15] H. Waeselynck, S. Behnia, "Towards a Framework for Testing B Models", LAAS Report No. 97225, June 1997.

[16] E.J. Weyuker, "On testing non-testable programs", *The Computer Journal*, Vol. 25, n° 4, 1982, pp. 465-470.