THE ROLE OF TESTING IN THE B FORMAL DEVELOPMENT PROCESS.

H. Waeselynck, J-L. Boulanger
INRETS–ESTAS
20, rue Élisée Reclus
59650 Villeneuve d'Ascq. FRANCE.

mail: {waeselync,boulang}@inrets.fr

Abstract

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

An on-going discussion in the community of B Users is the relevance of maintaining some classical verification steps of the design, e.g. unit and integration testing, for software that has followed such a formal development process. This paper presents our view as both assessors of automated transport systems and researchers in the field of software validation, and is aimed to provide sound arguments in favor of maintaining stringent testing policies.

After a brief presentation of the B method, we recall previous work that addresses the general problem of the limits of formal methods for ultra-high dependability. Then, the sequel of the paper is focused on the limits that hold in respect of the B method. A first problem is the level of maturity of both the method and the tools that support it. Then, the internal verifications (proofs) ensured by the method are described: it is shown that, although the method significantly contributes to fault avoidance, external verifications are still required at the various steps of the development process, whether inspections or tests. Finally, future direction for the search of an optimal verification strategy fitted to B is outlined.

Keywors : B method, formal development, software verification, safety-critical applications.

1 Introduction

Ultra-high dependability objectives are out of the scope of current evaluation approaches. As a result, the reliability level of a critical application may not be assessed quantitatively. Confidence on the acceptability of a system must be built all along the development process by combining stringent fault avoidance and fault tolerance techniques. *Formal methods* are good candidate to address fault avoidance: they can make significant contribution to both fault prevention and fault removal.

The formal *B* **method** is more and more used in France for the development of safety-critical software for railway systems. The first application of the method was made on SACEM [14], a cab signalling and speed control system operating on Paris RER line A. The safety part of the SACEM software was developed according to traditional methods, and proved a posteriori to be correct with respect to a *B* specification. More recent projects involve the complete formal development of software (from specification to code), as in the automatic train protection systems CTDC, KVB-KVS, METEOR and the interlocking system LST [5, 8]. The *B* method has also been used in other application fields: [9] reports on the development of a new component for the IBM Customer Information Control System (CICS); [20] presents a case study that is part of a marine diesel-engine monitoring system.

Since the *B* method covers all the software development process through a series of proved refinement steps, an on-going discussion in the community of *B* Users is the relevance of maintaining some classical verification steps of the design, e.g. unit and integration testing [1]. Three different opinions can be found:

- unit testing is no longer necessary and integration testing can be dramatically reduced [8];
- it is possible to remove or dramatically reduce these verification steps as soon as the tools that support the *B* method are certified;
- a stringent verification policy including all classical verification steps should be maintained anyway.

This paper is an attempt to provide sound arguments to support the third claim. It presents our view as both assessors of automated transport systems and researchers in the field of software validation.

Section 2 recalls previous work stating the limits of formal methods. Then, the discussion is focused on the *B* method, a brief overview of which is given in Section 3. Section 4 addresses the problem of the maturity of both the methods and the tools that support it. Section 5 describes the *B* development process and the internal verification that is ensured by the method: it is shown that additional verifications are still required at the various steps of the

[1]The terminology adopted in this paper is consistent with [17]. In particular, the word "verification" refers to the process of checking whether the system adheres to properties termed the verification conditions. Both static (e.g. inspections, proofs) and dynamic (e.g. testing) techniques are included in this definition.

development process, whether inspections or tests. Finally, future direction for the search of an optimal verification strategy fitted to B is outlined in section 6.

2 Previous work on the limits of Formal Methods

If formal methods are deemed necessary to achieve ultra-high dependability, previous work has allowed to identify their limits. Most of the material included in this section has been found in [6, 7, 13, 15].

Non-functional requirements (including dependability requirements) are not addressed by formal methods. In the sequel of this paper, the word "requirement" without any further comment will denote a "functional requirement".

Formal methods involve building formal models, and there are some intrinsic problems related to the activity of modeling. First, the model is necessarily an abstraction (simplification) of reality and has a limited scope. For example, the formal description of a piece of software takes into account neither the compiler used to product the executable code, nor the operating system, nor the hardware on which it is executed; it also makes assumptions on the behavior of any system component interacting with the piece of software. This idealized environment may not correspond to the real world. A second problem is that the model can be wrong with respect to its informal requirements. In essence, there is no way to prove that the informal (maybe ill-defined) user needs are correctly addressed by the formal specification of the service to deliver. Moreover, the modeling is performed by a human operator that is liable to fail.

It is worth noting that the limits mentioned above do not imply that formal models are useless. The validity and usefulness of such models should not be judged against an imaginary perfection but in comparison with the informal, ambiguous and not mathematically tractable models that would be used otherwise. Note also that the **problem of the external verification** [13] of a model with respect to its informal requirements does not arise solely at the specification level. As argued by [6], refinement (of an abstract specification) *will always require a certain degree of human input, admitting possibilities of human errors.*

A major argument in favor of the use of formal models is the possibility of establishing properties on them. Hence, it can be verified that a specification fulfils some safety requirements, or that a design is correct with respect to its specification (if both are expressed formally). Then the problem is that of the **validation of the validation** [17], or how to reach confidence in the methods and tools used in building confidence in the system. The formal system underlying the method may be wrong [2]. It is well-known that proofs may be faulty: the use of computerized tools significantly alleviates the problem, but there is still the issue of the confidence that can be placed on such tools.

[2]Fortunately, this theoretical problem is out of concern for the users of most formal methods.

In conclusion, formal methods must be used in combination with other traditional methods related to fault prevention, fault tolerance, fault removal and fault forecasting. As regards the issue of fault removal, [13] recalls that testing and proving are probably complementary, just as proofs and refutations are complementary. [6] advocates the maintaining of inspection, walk-throughs and testing policies, but suggests that testing would not need to be as exhaustive as in the case where formal methods had not been employed. To what extend could testing be reduced ? In the next sections, we will focus the discussion on the case of the *B* method.

3 Overview of the *B* Method

The *B* method due to J.R Abrial [2] is a formal method for the incremental development of specifications and their refinements down to an implementation. It is a model-based approach similar to **Z** [19] and **VDM** [16]. At each step of the *B* development some proof obligations are generated, enabling the verification of refinement as well that of abstract machine consistency.

The **abstract machine** [3] is the basic element of a *B* development. It encapsulates some state data and offers some operations. The description of an abstract machine is composed of three parts, the *declarative part* which describes the states and their properties, the *execution part* which introduces operations, and *composition clauses*. The Abstract Machine Notation (AMN) captures both specification, refinement concepts and implementation in one notation.

In the **declarative part**, the state is described with the set-theoretic model and the first order logic. *Constraints* on the formal parameters of the machine are introduced. Sets and constants are defined together with their *properties*. Variables are declared with their *initialization* and *invariant*. The invariant states the static laws that the data must obey whatever the operation that is applied to it.

x := E	Simple substitution
$S\|\|T$	Parallel substitution
skip	Empty substitution or no-op
$P \mid S$	Preconditionning
$S[]T$	Bounded choice
$P \implies S$	Guarding
$@x.S$	Unbounded choice

x denotes a variable, E is a set theoretical expression,
P denotes a predicate, S and T denote generalised substitutions.

Figure 1: A subset of Generalised Substitutions

The **execution part** introduces some operations which are described under the form of a pre-condition and an action. These operations provide the interface for the outside world to the state variables encapsulated within the abstract machine. Operations are described by syntactic structures which are interpreted in the *generalized substitution language* [4], an extension of earlier work by Dijkstra [11]. The generalised substitutions (see figure 1) are predicate transformers. $[S]P$ denotes the result of applying substitution S to the predicate P.

`IF Q THEN S ELSE T END` is an example of action described in the body of an operation. It is interpreted in the generalised substitution language as the guarded choice `Q => S  [] not(Q) =>T`.

The **composition clause** part allows to introduce relationships between abstract machines (SEES, USES, INCLUDES, IMPORTS, EXTENDS) according to various visibility rules on the encapsulated states and operations.

```
MACHINE
   STACK (max_object)
  CONSTRAINTS
     max_object : NAT1
  SEES
     OBJECT
  VARIABLES
     stack
  INVARIANT
     stack  : seq(Object) &
     size(stack) <= max_object
  INITIALISATION
     stack := <>

  OPERATIONS

     PUSH (XX) =
       PRE
         XX : Object & size(stack) < max_object
       THEN stack := stack <- XX END;

     XX <-- POP =
       PRE
         size(stack) > 0
       THEN XX,stack   := last(stack), front(stack) END
 END
```

Figure 2: Example of abstract machine

An example of abstract machine is given in figure 2 where a fairly simple *STACK* is presented. The declarative part defines one formal parameter with its constraint, and one variable with its typing invariant and its initialization. Two operations provide access to the encapsulated variable. The SEES composition clause allows to "see" the set *Object* defined elsewhere.

Each abstract machine involves a **development chain** including the top level, it refinement(s) and its implementation, the latter being easily translated into a programming language. Although the AMN covers the whole chain, a few syntactic structures cannot be used at any level. For example, the non-deterministic choice `ANY x WHERE P THEN S END` cannot be found in an implementation while the loop structure `WHILE B DO S END` is not accepted at the abstract top level. At each step of the *B* development, a collection of proof obligations is introduced, which will be presented in section 5.

It is worth noting that the links established by the composition clauses provide access to the top level of the referred abstract machines: machines can be refined, implemented and proved separately, which introduces modularity in the *B* development.

4 Degree of maturity of the *B* Method and Tools

Let us go back to the problem of the *validation of the validation* (the confidence in the validation methods and tools), and examine it in relation to the degree of maturity of both the *B* method and the tools that support it. Part of the discussion has benefited from the material provided by [10].

4.1 Maturity of the *B* method

The emergence of the *B* method is recent. Its first significant uses for industrial projects were carried out at British Petroleum and Gec Alsthom during the late eighties. At that time, the development of tools built on a formula manipulator, the *B-Tool*, helped to consolidate the notation and semantics. The *B* method has now been used for several industrial developments, and two commercialized tools are currently available, the *B-Toolkit* (July 1994) and the `AtelierB` (January 1995). As yet few material has been published in the literature, but several books are expected to appear in 1995, including the B Book of Abrial [2] that will give a detailed presentation of the theoretical aspects of the method.

The *B* syntax is not yet fully stabilized, minor evolutions being planned owing to feedback from the tool users. Also, slight differences can be found between the English and French schools (corresponding to the two existing tools). But it must be stressed that the bulk of the *B* notation and semantics is already well-defined. More importantly, the method relies on sound theoretical foundations. The correctness criteria that guide the generation of proof obligations have been established by previous work on model-based methods (e.g. [16, 18]). Hence, in spite of the relative novelty of the *B* method, confidence can be placed on the internal verification steps that it prescribes.

Whereas the method itself can be considered as mature on a theoretical viewpoint, there is currently no agreed *methodology* for carrying out a *B* development process, although some hints can be found in [1]. It is worth noting that the same problem arises with any new modeling method. As more and more experience is gained on *B* practice some general procedures emerge, but the know-how has been little disseminated outside the industrial companies. This may cause problem because, as will be argued, an inadequate approach to the modeling in *B* will have an indirect impact on the reliability of the resulting software.

Part of the problem is to **choose the right level of abstraction at each development step**. The *B* notation covers a large spectrum, from top level machines to implementations, and there are very few limitations concerning the use of the various syntactic elements at the different levels. Starting from an algorithmic, detailed description of *B* machines is allowed by the syntax, but would be poor practice. On the other hand, from what we have seen from successful industrial projects, there is no need to carry out a large number of refinement steps: there is typically at most one refinement (if any) between a top level machine and its implementation, the details being captured by decomposing the problem into smaller pieces (e.g. imported machines at the implementation level) rather than by having large complex machines.

Another serious problem is**how to structure the development**. The *B* method provides us with a lot of means to establish links between the machines (clauses INCLUDES, SEES, USES, IMPORTS ...), each involving different visibility rules. There are some stringent constraints that can be troublesome for people used to structured analysis methods [3]: for example, an operation cannot be called from within an operation of the same machine; at most one operation of an included machine can be called from within an operation of the including machine [2]. Indeed designing the proper structure for a given application is far from trivial. The inexperienced developer may ends up by building a large number of machines with a complex network of relations between them. Not only it will be difficult to understand the context of a given machine according to the visibility rules, but there is also the risk that some important properties (e.g. invariants) of the application cannot be expressed because the relevant parts are inadequately split into the body of different machines.

The two problems mentioned above, abstraction and structuring, may result in having the *B* developer lose control of its work. Then the probability of introducing faults increases, while the external verification is impeded by the obscurity of the development. Moreover, a "dirty" development may also result in having complex proof obligations generated, most of them not being proved automatically, hence requiring human intervention. Once again, this is not in favor of fault avoidance.

To conclude, the benefit that can be drawn from the *B* method depends on the way it is applied. In particular, the inexperienced *B* developer should not feel a false sense of security because he is using a *formal* method. Testing and inspections are necessary to provide him with feedback on the quality of the development, and all the more so as no defined development methodology is currently available.

[3]Such constraints are necessary to protect the encapsulated state of a machine and avoid breaking the invariant by side effects.

4.2 Maturity of the *B* tools

As already mentioned, there are currently two recent commercialized tools supporting the *B* method: the `AtelierB` of Digilog and the *B-Toolkit* of BCore. The facilities provided by each of them are summarized in figure 3.

	AtelierB	B-Toolkit
Syntax-checker, Type-checker	yes	yes
Proof Obligation Generator Automatic Prover Interactive Prover	yes	yes
Library of basic machines	yes	yes
Translator	C, ADA(*)	C,PL/X, ADA(*)
Animator	(*)	yes
Graphical Interface	yes	(*)
Formatter	LaTeX, Interleaf(*)	LaTeX

(*) : to be supplied soon.

Figure 3: Facilities offered by the *B* tools

A tool session can be described as follows: once the source of a machine (which can be either a top level machine, a refinement, or an implementation) has passed the syntactic and type checks, a collection of proof obligations (POs) are generated in conformity with the *B* method. Then the POs are processed by the automatic prover that works in conjunction with a library of mathematical rules. Some of the POs will not be discharged: the interactive prover allows the user to create tactics or to add new mathematical rules for their proof. Once each of the remaining POs has been successfully processed, the automatic prover must be invoked again on the whole set of POs in order to verify that the tactics and rules added by the user do not invalidate the previous proofs. Finally, if the target *B* machine is an implementation, translation into a programming language can be launched.

As already said, a few slight evolutions of the supported *B* syntax are planned for the tools; but the major evolution that is required by the users mainly concerns the power of the automatic provers: whatever the *B* tool, any real-size application currently requires a lot of human intervention to perform the prescribed proofs.

The validation of both tools has involved intensive testing, using a large set of *B* sources as a benchmark. Also, both of them have been subjected to beta-tests in a number of external sites. However, it must be stressed that none of them can be considered yet as *certified*. To our knowledge, work is currently being carried out in France to strengthen the verification of 1) the library of mathematical rules used by the prover and 2) the structure of the generated proofs. It is worth noting that this work will be valuable not only to validate the subsequent versions of the prover, but also to provide the user with tools to verify the set of rules that are manually added. Finally, there is the issue of the translation of *B* implementations into a programming language: this step is out of the scope of the *B* method and, strictly speaking, there is currently no *formal* verification of it.

To conclude this section we claim that, for safety-critical applications, a stringent verification policy should be maintained at least because of the lack of maturity of a development methodology suited to *B*, and because none of the tools that support the method has been certified yet. In what follows, we will examine the internal verification mechanisms of *B*, and focus our argumentation on the problem of the *external verification*.

5 Verification of the Formal Development

The common picture associated with formal methods and refinement steps is the one described in 4: an abstract model is designed to capture the informal requirements; then a series of refinement steps is undertaken down to an implementation. Since each step is proved with respect to the previous one, the problem of the external verification is often focused on the verification of the abstract specification with respect to its informal requirements. Actually, we will see that the *B* development follows a different scheme so that external verification is required *all along* the development process.

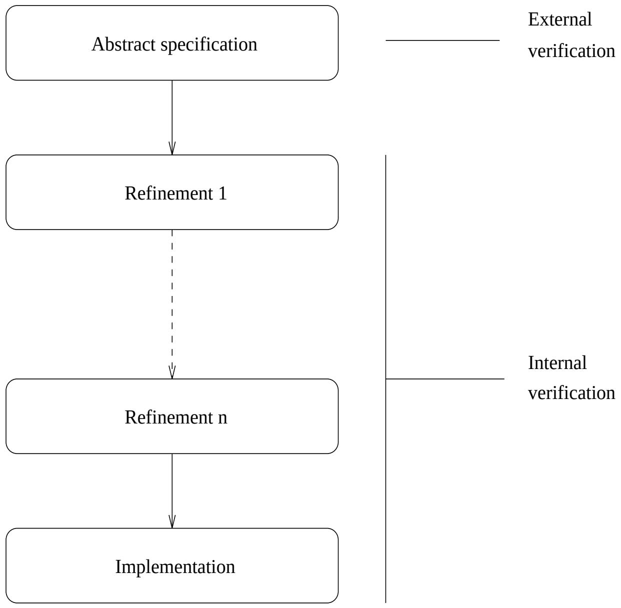

Figure 4: Common picture associated with a formal development

5.1 *B* development scheme

The *B* development scheme is described in Figure 5. It is in conformity with the approaches followed in [1, 9].

As can be seen, the backbone of the development structure is provided by the IMPORTS clauses that are introduced at the implementation level of the various development chains. The IMPORTS clause allows us to decompose a problem into smaller pieces, and may be related to the notion of layered architecture where each layer is using the service of the lower layer. In the example, the implementation of the *Main* machine is using the service of two machines imported from the lower layer: the algorithms described in `Main_1.imp` may refer to the operations of *FCT1* and *FCT2*. *FCT1* and *FCT2* are in turn refined down to an implementation that may import subsidiary machines, and so on.

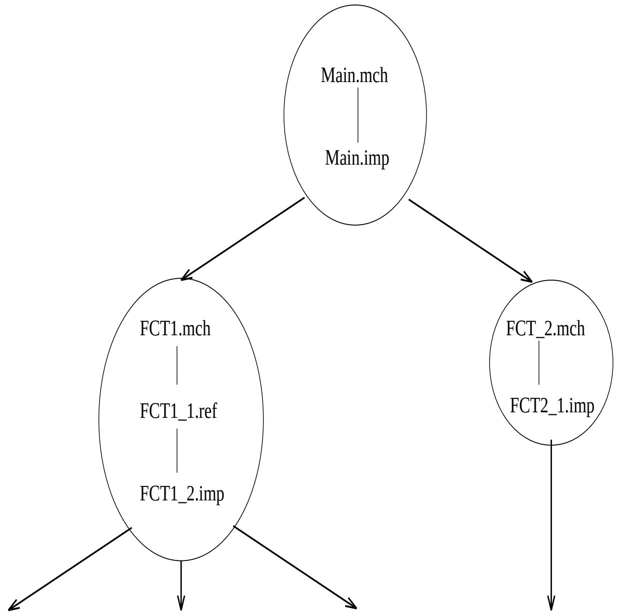

Figure 5: The *B* development scheme

The development chain of one machine is more than introducing algorithmic details. New functional features may also be stated: a minimal top level machine may contain only the signature of a set of operations, their body being described at the lower levels. This involves few refinement steps: as already mentioned, the complexity is captured by decomposing the problem into smaller pieces rather than by having "large" abstract machines. Also, we have described a top-down decomposition, but part of the development may be bottom-up as well, especially if existing machines are to be reused from other projects. Finally, there is another useful clause not shown in the example, the SEES clause, which can be introduced at the top level of a machine to express read-only relationships. Theoretically, the IMPORTS and SEES clauses are sufficient to structure any application.

It is worth noting that the development scheme described in Figure 5 is quite different from the one in Figure 4. **The specification and design tasks are no more separated:** *both* the details of the problem *and* the details of the implementation are gradually introduced, by making continuous backward and forward moves between the abstract and concrete world.

To conclude:

- new machines are created all along the modeling process;
- the dependency links mainly involve the implementation level of one machine and the top level of the imported machines (layered logical structure);
- the details of the problem are taken into account both by the refinement process and by the composition/decomposition of machines.

Such an approach is preferable to the one consisting in writing a (flat) complete specification, and *then* starting the refinement. Trying to capture all the required functional features at the beginning of the process would amount to create a large number of machines at once and to express the dependency links between their top levels (clauses INCLUDES, USES, ...). This would bring us to the methodological problems described in 4.1: lack of abstraction, obscurity, complex network of relations between the machines.

5.2 Internal verification prescribed by the *B* method

For a given machine, three kinds of verification are prescribed by the method: syntax checking, type checking and proof obligations.

Syntactic and type analysis The syntactic analysis ensures that the source of an abstract machine satisfies the syntactic rules of the AMN. Then, type checking is carried out. This stage is important because no attempt will be made to prove a predicate involving constructs that have not been checked for type consistency. Note that it is possible to determine the type of any expression because the set-theoretic model used in AMN is a simplification of the classical set theory.

The type checking allows to reveal faults like those related to missing or incorrect declarations, type inconsistency within an expression, inconsistency of the signature of an operation with the signature of its refined version, violation of the visibility rules induced by the composition clause part.

Proof obligations In the *B* development, the proofs accompany the construction of software. Each time an abstract machine is defined or modified, there are proof obligations related to its mathematical consistency; if the machine is a refinement or an implementation, there are also proofs of its correctness with respect to the previous steps of the development chain. The *B* tools allow to generate automatically the POs for each abstract machine. Generally speaking, the POs will be all the more complex as concrete details are introduced.

All POs contain in their hypotheses some information which describes the context of the abstract machine. The context contains the constraints on formal parameters and the properties of sets and constants. It contains also the assumption that all used sets are finite, not empty and that their elements are distinct. We note $< Context >$ this part of hypotheses. The composition clause part can add information to the context, the invariant and the initialization according to the visibility rules.

Checking the **mathematical consistency** involves two kinds of PO. First, it must be proved that the model is not empty. This is done by requiring that the initialization establishes the invariant:

$$< Context > \Longrightarrow [Initialisation]Invariant \tag{1}$$

This PO relates to the feasibility of initialization, but it does not amount to an executable initialization (the initialization may be done by using a non-deterministic substitution). Second, it must be proved that each operation preserves the invariant. In this case for each operation OP defining a substitution S under the pre-conditioning Q there is a PO of the following form:

$$< Context > \wedge Invariant \implies Q \wedge [S]Invariant \tag{2}$$

Checking the **correctness of a refinement** with respect to the previous steps of the development chain involves checking that its initialization and operations preserve the semantics of their corresponding more abstract versions. The POs generated for the nth refinement are described by Equation 3 for initialization and by Equation 4 applied to each operation.

$$< Context > \Longrightarrow [Init_n]\neg[Init_{n-1}]\neg I_n \tag{3}$$

$$< Context > I_1 \wedge I_2 ... \wedge I_n \wedge Q_1 \implies Q_n \wedge [[u_n := u_{n-1}]S_n]\neg[S_{n-1}]\neg(I_n \wedge u_n = u_{n-1}) \tag{4}$$

Where $Init_i, I_i, Q_i, S_i, u_i$ are respectively the initialization, the invariant, the pre-condition of the operation, the action of the operation, the formal parameters of the operation at the i^{th} refinement.

It is worth noting that there is no *existence* proof for the predicates involved in the invariant, pre-conditions of operations, constraints or properties: *constructive* proofs are performed when existence is required. The verification of the existence of variables which satisfy the invariant is indirectly introduced by the POs 1 and 3. The existence of the formal parameters which validate the pre-condition of an operation is indirectly introduced by the POs 2 and 4. The two other cases related to constraints and properties are described below.

The existence of formal parameters which validate the constraints of a target machine is verified when this machine is included or imported in another one. Then the machine which includes or imports the parametrized machine must instantiate the formal parameters, and the PO 5 is generated to verify that these actual parameters satisfy the constraints clause:

$$< Context > \Longrightarrow [Formal\_parameter := Actual\_parameter](A \wedge Constraints) \tag{5}$$

Where the predicate A states that the sets passed as parameters are finite and not empty and Constraints is the clause of the Declarative part which defines some constraints on the formal parameters.

The valuation of sets and constants may be done either at the top level or at the implementation level (leading to deferred sets and constants). The PO 6 is generated at the implementation level to verify that the properties hold in respect of the chosen values:

$$< Subset\_of\_Context > \Longrightarrow [DeferredSET, DeferredVAR := Values]Properties \quad (6)$$

We recall that Properties is the clause of the Declarative part of abstract machine which defines some constraints on local sets and constants deferred or not. $< Subset\_of\_Context >$ is the part of the context that is used to construct the values.

Having constructive proofs implies that some checks are delayed and performed only at the implementation level. In the case of a parametrized machine, some checks are delayed until the machine is included or imported in another development chain.

Finally, it can be seen from the previous equations that the number of predicates involved in the POs grows with the number of refinement steps, and with the number of links introduced by the composition part.

5.3 The external verification problem

```
uu,vv,ww <-- Sort_3_NAT( xx, yy, zz ) =
    PRE
         xx :NAT & yy :NAT & zz:NAT
    THEN
        ANY
              ua,va,wa
        WHERE
              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
```

Figure 6: Inadequate description of a sorting problem

The internal verification prescribed by the *B* method significantly contributes to fault avoidance. However, there is still the problem of the *external* verification. Let us recall that new functional features are gradually introduced all along the development process, either by refinement of an abstract machine or by composition with other machines (see 5.1). Hence, external verification must accompany the whole process. Note that this should be easier than having to verify at once a large, flat specification because it should be possible to take advantage of the modularity of the development.

External verification aims at tracking down faults originating from a **misunderstanding of the requirements**, or from the **failure to express adequately an understood requirement**. Figure 6 gives an example of the latter case. It is an attempt to describe a simple operation that sorts three integers. The developer has inadequately used the concept of set, which can be revealed by considering the case where two inputs are equal. Let us take $xx = yy = 2$ and $zz = 1$, which implies that $\{xx, yy, zz\} = \{1, 2\}$. Then both $uu = vv = 1$, $ww = 2$ and $uu = 1$, $vv = ww = 2$ are accepted as outputs. Figure 6 is typically an underspecification problem. In the general case, there is no way to prove that a formal model satisfies its *intended* behavior. Indeed, there are several examples of published pieces of specification in which faults are found afterwards, although they may have seemed simple and self-explanatory at a first sight.

```
MACHINE
     TIMEOUT
CONSTANTS
     TMAX
PROPERTIES
     TMAX : NAT
VARIABLES
     alarm ,timer
INVARIANT
     alarm: BOOL
&    timer : NAT
&  ((timer > TMAX) => (alarm = TRUE))
INITIALISATION
     alarm,timer := TRUE,0
OPERATIONS
     INCR =
     BEGIN
       IF timer < TMAX
       THEN timer := timer +1
       ELSE timer := timer +1 || alarm:= TRUE
       END
     END
     RESET = ...
END
```

Figure 7: A timer manager

Ideally, the choices made during the *B* development should be driven by a preliminary functional analysis aiming to identify a set of important properties of the application. Then software development proceeds to ensure that these properties are satisfied *by construction*. Apart from the fact that some critical properties may have been ignored, it results from what precedes that faults can be introduced when expressing the properties in the formal development. Figure 7 presents a simple timer manager. A local invariant states that an alarm must be issued if the timer exceeds the value TMAX: the *B* method ensures that this property holds whatever the choices made in the initialization part and in the body of the operations.

However, it can be seen that nothing prevents the alarm from being always TRUE. Another problem is to verify that the timer is incremented or reset under the appropriate conditions: this would have to be done in the machine that invokes the corresponding operations. From this simple example, it can be concluded that external verification must be performed not only at the local level (verification of the development chain of a target machine) but also at the level of the machines that use the local service (verification of the interactions).

External verification also involves the **handling of the proof obligations that are not discharged** by the automatic prover. Note that a first problem is to distinguish the cases where the prover has been insufficient among the cases where a fault has been exposed. The diagnosis is far from trivial because the expression of a PO is often complex, and none of the *B* tools allows to trace precisely the piece of *B* machine from which it originates. Hence it is not unlikely that the wrong POs be selected. Then, the user adds manually mathematical rules to help the prover to proceed with the proofs. These rules may belong to one of the two following categories :

- *general rules* that can be related to the mathematical theory underlying the *B* method. For example, a current weakness of the *B* tools is the handling of proofs involving arithmetic expressions. For positive integers, rules like: $u \leq v \implies u * u \leq v * v$ may be introduced to compensate for this weakness.

- *application-specific rules* that have to be interpreted in the framework of the application. For example, the rule $pp = TRUE \implies not\_pp = FALSE$ holds if the Boolean variable *not_pp* always contains the negation of *pp*. It is obvious that the user may introduce false rules that allow him to prove false lemmas, the problem arising most acutely with application-specific rules.

It is obvious that the user may introduce false rules that allow him to prove false lemmas, the problem arising most acutely with application-specific rules.

6 Conclusion and Future work

Quoting from [20]:*"formal methods, and in particular the B-methodology, should not be viewed as a replacement for other types of verification and testing but should be seen as a means by which the quality and reliability of a system can be improved"*. We fully subscribe to this opinion. We argue that the *B* method should be supplemented by both static (e.g. inspections) and dynamic (e.g. testing) verification techniques for four main reasons:

- to expose methodological problems with the use of *B* that could have an indirect impact on the reliability of the resulting software;
- to introduce redundancy with respect to the internal verification supported by recently commercialized tools that have not yet been certified;
- to supply external verification of the formal development with respect to the informal requirements;
- to verify the user rules that are added to proceed with the proofs.

The benefit that can be gained from using the *B* method should not be set in terms of the amount of verification to be performed, but in terms of the amount of *rework* that is needed for fault removal, and in terms of the *confidence* that can be placed on critical software.

Defining **inspection protocols for a *B* development** would require that classical checklists be accommodated to take into account the peculiarities of the method. For example, it would be useless to verify that all variables are declared: it is ensured by construction. Of greater interest would be the examination of how the critical properties of the application are taken into account in the invariant of the machines; then, any operation necessarily preserves the invariant. [12] emphasizes that inspections require that the development be made *with the criterion of inspectability in mind*. In the case of *B*, this means that work is to be done 1) to define the kind of functional analysis to be conducted before the development in order to facilitate traceability checks of the requirements (identification of the state variables, of the equations of the system, see [1]) and 2) to improve the readability of the *B* source (definition of good "programming" rules, rules to comment the source). In addition to *B* source inspections, **inspections of the generated proof obligations** would foster better understanding of what is stated in the body of the machines. Also, complex proof obligations could be an indicator of methodological problems.

Testing is necessary to verify the dynamics of the software. As was argued in 5.3, local verification of a development chain as well as the verification of the interaction of machines are required. Hence, we believe that all classical steps, namely **unit**, **integration and global testing** should be maintained. However, unit testing could be entirely replaced by *B* **machine animation** if the translator into programming languages, and the associated compilers, could be relied on. Note that testing as well as *B* animation raises the *oracle problem*, or *how to determine the correct output results a program should return in response to given input data*. Once again, we stress the importance of the functional analysis that precedes the *B* development.

Another role for testing could be the **verification of the application-specific rules** introduced by the user, the general rules being preferably verified by proofs. In this case, the oracle is no more a problem since the target rules form the properties to verify. The same approach may be followed to verify afterwards some critical properties that have not been taken into account explicitly in the design. For example, when performing the assessment of a new transport system, the INRETS experts may ask the supplier to verify additional safety properties. The design of test inputs specifically focused on a given property is currently under study. Our research on this issue will be supported by an experimental work conducted on a function extracted from an automated train protection system developed in *B*.

Acknowledgements

This work has been stimulated by the discussions conducted in the framework of the *B* User Group (BUG); this paper expresses the opinion of the authors, not of the group. We wish to thank Jean-Raymond Abrial and François Baranowski for their suggestive comments during the preparation of this paper.

Contents

1 Introduction **2**

2 Previous work on the limits of Formal Methods **3**

3 Overview of the *B* Method **4**

4 Degree of maturity of the *B* Method and Tools **6**
4.1 Maturity of the *B* method 6
4.2 Maturity of the *B* tools 8

5 Verification of the Formal Development **9**
5.1 *B* development scheme 9
5.2 Internal verification prescribed by the *B* method 11
5.3 The external verification problem 13

6 Conclusion and Future work **15**

References

[1] J-R. Abrial. A boiler case study. private communication, 1995.

[2] Jean-Raymond Abrial. The b-book, 1993.

[3] J.R. Abrial. On constructing large software systems. In *Algorithms, Software, Architecture. Information Processing 92. IFIP 12th World Computer Congress*, volume A-12, pages 103–12, 7-11 Sept. 1992.

[4] J.R. Abrial, M.K.O. Lee, D.S. Neilson, P.N. Scharbach, and I.H. Sorensen. The b-method (software development). In W.J. Prehn, S.; Toetenel, editor, *VDM 91. Formal Software Development Methods. 4th International Symposium of VDM Europe Proceedings.*, volume 2, pages 398–405. BP Res., Sunbury Res. Centre, Sunbury-on-Thames, UK, Springer-Verlag, Berlin, Germany, October 1991.

[5] P. Behm. Application d'une méthode formelle aux logiciels sécuritaires ferroviaires. In *Atelier Logiciel Temps Réel, 6ème Journées Internationales du Génie Logiciel*, 17-19 novembre 1993.

[6] J. Bowen and M. Hinchey. Ten commandments of formal methods: some guidelines for successful use. Technical Report No 350, University of Cambridge Computer Laboratory, September 1995. To appear in IEEE Computer, March 1995.

[7] A. Cohn. The notion of proof in hardware verification. *Journal of Automated Reasoning*, vol. 5:pp. 127–139, 1989.

[8] B. Dehbonei and F. Mejia. Formal development of software in railways safety critical systems. In B. T.K.S. Murthy, C.A. Mellitt, G. Brebbia, and S. Sciutto, editors, *Railway Operations*, volume 2, pages 213–219. COMPRAIL94, Computational Mechanics Publications, 1994.

[9] J. Dick. *Applications of formal methods*. Prentice Hall, International Series in Computer Science, 1995.

[10] J. Dick. An assessment of the maturity of the b technologies. private communication, 1995.

[11] E.W. Dijkstra. *A Disciple of Programming*. Prentice Hall, 1976.

[12] M.H. Van Emden. Structured inspections of code. *Journal of Software Testing, Verification and Reliability*, vol. 2:pp. 133–153, 1992.

[13] M-C. Gaudel. Advantages and limits of formal approaches for ultra-high dependability. In *in Proc. International Workshop on Software Specification and Design (IWSSD'91)*, October 1991.

[14] G. Guihot and C. Hennebert. Sacem software validation. in Proc. 12th IEEE- ACM International Conference on Software Engineering, March 1990.

[15] A. Hall. Seven myths of formal methods. *IEEE Software*, pages pp.11–19, September 1990.

[16] Cliff B. Jones. *Systematic Software Development using VDM.* Prentice Hall International; Second Edition, 1990.

[17] J-C. Laprie. *Dependability: basic concepts and terminology.* Springer Verlag, Vienna, 1992.

[18] C. Morgan. *Deriving programs from specifications.* Prentice Hall International, 1990.

[19] J. M. Spivey. *The Z notation- a reference Manual.* Prentice Hall International, 1989.

[20] A. Storey. A specification case study using the b-methodology. *Software Testing, Verification and Reliability*, 2(4):187–202, December 1992.