Revisiting B Language syntax
Submission to the FM99 - B mini-track

Georges Mariano[1], Jean-Louis Boulanger[2], and Bruno Tatibouët[3]

[1] INRETS[†],
Rue Élisée-Reclus, 59550 Villeneuve d'Ascq CEDEX France
mariano@inrets.fr
[2] CEDRIC–CNAM[‡],
55, rue de Turbigo 75003 Paris
boulang@cnam.fr
[3] LIB[§],
16 route de Gray, 25030 Besançon CEDEX, France,
Tel +33 (0)3 81 66 64 55, Fax +33 (0)3 81 66 64 50,
tati@lib.univ-fcomte.fr

Abstract. Building auxiliary tools (quality measures [6,14], dedicated editors[1] and parsers[2]) implies the availability of a model describing the formal text as a grammar [13,20] [4].
During the development of this modeling, we highlighted anomalies of notations. These problems increase the difficulty of setting tools dedicated to the automatic/static and syntax based analysis of specifications. These problems, and their effects on the use of the B method, are characterized in section 1.
Section 2 describe the solution that we have built upon necessary modifications of the B language. These solutions allow us to define a (formal) grammar covering the whole of the B notation, altough slightly modified.
Section 3 enumerates the various prospects for this work, with regard to B language and with regard to the possible extensions of our work.
In conclusion, section 4, by considering in a more total way the described problems, we will evaluate the impact of our results from a B standardization point of view and also describe some futur works.

[†] Institut National de Recherche sur les Transports et leur Sécurité
[‡] Centre d'Études et de Recherche en Informatique du CNAM
[§] Laboratoire d'Informatique de Besançon
[1] http://lib.univ-fcomte.fr/PEOPLE/tatibouet/EditeurB.html
[2] http://perso-wanadoo.fr/bruno.tatibouet/BParser_en.html

Table of Contents

Introduction

The *B* method [1], developped by Jean-Raymond Abrial, is a 'model-based' formal method like *Z* [18] or *VDM* [9] but, unlike others, *B* allows an incremental process development (also called refinement process [16]) starting from abstract specification to (automatically generated) code.

In this paper we are particularly concerned by the fact that this process is covered by *one and only one language* : the Abstract Machine Notation (AMN). In the following we will use the name 'B language'. We suppose in the following that the reader is rather familiar with the *B* language.

All formal notation is confronted by two imperatives which can be contradictory :

- the first one consists in having a notation flexible enough, to prevent the user to be putted off by unnecessary constraints (for example $a, b, c \in \mathbb{N}$ is more practical (but rather incorrect) than $a, b, c \in \mathbb{N} \times \mathbb{N} \times \mathbb{N}$ which is better than $a \in \mathbb{N} \wedge b \in \mathbb{N} \wedge c \in \mathbb{N}$),
- the second one in having a syntax allowing automatic tool processing.

The method B uses a notation which associate ASCII text to to each mathematical operators (for example the mathematical operator $\mapsto$ is written `|->`). This notation has the advantage to be easily manipulated by any tools as *electronic mail*, *editor* or *programming languages*. The disadvantages of this type of notation are known :

- a readability reserved to practitioners of notations,
- syntax analysis more delicate

The purpose of this paper is to show that, at the present time, the *B* tools are based on an ASCII notation, also called concrete notation, which exhibits some problems.

Operators overloading First, a same textual representation may be associated for different operators; this is what we will call operator overloading. This is the case for the symbol $\boxed{\texttt{;}}$ which represents the end of some declarations, the sequential composition of substitutions and the composition of relations. Also, the symbol $\boxed{\texttt{||}}$ represents the parallel product of relations and the parallel composition of substitutions. There are other examples of overloadings but the induced ambiguities can be removed during the processing of type-checking (sets difference and integers difference, multiplication of integers and cartesian product of sets).

Textual expansion of the DEFINITIONS clause Second, to make easier the writing of large and/or complex specifications, the mechanism of *definitions* has been integrated in the *B* notation and in *B* specifications processing.

A *definition* allow to nominate any text, possibly with parameters, and to use it again, in any point of the formal text, by a simple recalling of the associated name. Today, the definitions are textually expanded by a similar technique to the pre-processing (cpp) before any other processing by the *B* tools.

The mandatory use of this textual expansion infer two essential problems : first of all, it is impossible to impose rules of a good use of the definitions and secondly, the expansion not controled in the formal text may infer some problems bound to the priorities of the different implicated operators.

The following section is dedicated to the exposure of the consequences induced by these two problems. We start by considering the definitions (section 1.1) and then the overloaded operators (section 1.2).

1 The problems

To better understand the interesting aspects of the use of the definitions in making easier the writing of the formal specifications, we show some examples of definitions in *B* , then we will set up a "typology" of the different way of using the definitions that we can meet; We share this typology in two : first of all, the forms encountered the more frequently and so that must be taken into account imperatively because they correspond to a need and forms more rarely encountered which seem abusive for us.

1.1 The DEFINITIONS clause

A *definition* is the association between *any text* and a name which will allow to use it in the current machine, refinement or implementation. This association is declared through the clause DEFINITIONS . Thereafter are three examples of definitions :

```
DEFINITIONS
    xx == distance := 3 ;
    yy(ee,dd) == ee + bb ;
    Safety == distance <= 100
```

Regarding our experience, these 3 examples show some possible and typical use of definitions : 1) a substitution, 2) an expression with parameters, 3) a predicate.

Example 1 : Faulty computations Here is a *B* machine, using the definitions above :

```
MACHINE
    MyFirstVerySimpleExample
DEFINITIONS
    ...
VARIABLES
    distance
INVARIANT
    distance ∈ ℕ
INITIALISATION
    xx
OPERATIONS
    nn ⟵ MyOperation(dd) =
        PRE dd ∈ ℕ ∧ Safety THEN
            nn := yy(distance, 100) * 10
        END
END
```

After textual expansion of the definitions calls, operation's body is now:

```
    nn ⟵ MyOperation(dd) =
        PRE dd ∈ ℕ ∧ distance ≤ 100 THEN
            nn := distance + 100 * 10
        END
```

Even regarding this example where the use of the definitions seems reasonable, a simple problem appears : *regarding the priority of the operators*, the estimation of $(distance + 100) \times 10$ appears as $distance + (100 \times 10)$ which was not the purpose of the developper. Further more, it is not sure that this mistake will be catch during the proof task. An easy way to solve this problem is to enclosed the definition text in parenthesis. However, the real problem remains, how can we detect such faulty computation ?

Without any syntactic/semantic consistance, definitions cannot take into account subtil concepts like operator's priority (the problem we just have described) and the more classical problem of variable capture (i.e scope of identifiers). Thus, and rather surprisingly, it is the responsability of the specifier to be safe against these threads. Here we may recall that formal language are designed to avoid these problems...

Example 2 : Unreadable machine The following example is a correct B machine, that is to say, a «formal text» accepted by the tools supporting the method. The reader will easely admit that it is the contrary of all the welldone rules of style designed to improve the readability and the understanding of a specification or a program.

```
MACHINE
    abracadabra
DEFINITIONS
    operation == header = body ;
    header == param_out ⟵ op_name param_in ;
    param_out == list_var_1 ;
    list_var_1 == v1,v2 ;
    op_name == foo ;
    param_in == ( list_var_2 ) ;
    list_var_2 == v3,v4 ;
    body == PRE PREDICATE THEN substitution END ;
    predicate == list_var_2 : list_type ;
    list_type == ℕ × ℕ ;
    substitution == list_var_1 := list_value ;
    list_value == list_var_2
OPERATIONS
    operation
END
```

These outstanding writings are possible due to the absence, almost total, of limits of what we can call the "body" of a definition (right hand side of $\boxed{\texttt{==}}$).

One may reply that specifiers will never use this kind of writing. Maybe, but it is still surprising, that the bases of a formal approach, namely rigourous design and readability of the specifications, can be so easily reduced to nothing without any control.

Lists of variables, constants... The shape $\boxed{vars == var_1, ..., var_n}$ allows to save the capture of a name from a list of variables. Here is an example of such a definition and a possible use :

DEFINITIONS
    $Ns == N1, N2, N3$
INVARIANT
    $Ns \in \mathbb{N} \times \mathbb{N} \times \mathbb{N}$
INITIALISATION
    $Ns := 0, 0, 0$

However, the benefit of such a definition is only possible in the case where, in the same time, the number of "occurrences" is important and the whole of variables will not be modified. Since this use is not (directly) related with readability or easy maintenance, we may consider it as abusive use of definitions.

Predicates and properties The shape $\boxed{prop_i == p_1 \wedge \ldots \wedge p_n}$ (where p_i are predicates) is much more interesting because it allows us to associate a symbolic label with a `well-formed` sentence of the language.

What do we mean by '`well-formed` sentence' ? We consider as '`well-formed` sentence' any formula of the language which may have an interpretation in itself. This is exactly the use mentioned in [12, p.24]. Kevin Lano puts restrictions on what can be a definition when he states that a definition def_i may have the shape $\boxed{data_i == val_i}$. The point which is important here is that the right part can be 'valued', thus it have to be a semantic value.

The latter use is much more beneficial in the field of the legibility and the maintainability of the formal text. The essential contribution comes owing to the fact that the "cohesion" induced by this type of abbreviation is not reduced to a syntactic facility but corresponds to the concretization of a concept having a significant meaning in the spirit of the specifior; this concept can be a fundamental property of the system or a generic mathematical expression (by using the possibilities of parameterized definitions).

Moreover, by a generalized use of this technique, it is possible to define a hierarchical construction connecting such definitions. This step would be based on the " definition " of primitive properties, then the construction of complex properties worked out by agglomerations of the preceding ones.

In fact, the DEFINITIONS clause was introduced to fill the need for a simple way to put names on, or to simply hide, (complicated) formulas. This is like procedure or local functions in classical programming languages excepting that, like 'macros', definitions do not have any syntactic existence and so we cannot attached them to any semantic property.

1.2 Operators overloading

A few operators in the B language have different meanings depending the context where they may appear. To clearly expose the consequence of this fact, we will submit a small example (see figure 1) to the available B tools.

The parallel operator ("||") denotes, on one hand, the parallel composition of relations and, on the other hand, the parallel composition of substitutions. We can say that parallel operator appears, with the same external representation, in the sub-language dealing with expressions and in the sub-language dealing with substitutions. This is what we (informally) call 'operator overloading'.

In the following machine (figure 1), which is correct with respect to the rules stated in [19], $R1$ and $R2$ are relations, and for human reader there is no confusion : xx is a relation resulting from the relational product $R1 \mid\mid R2$:

Unfortunately, this is not B tools interpretation as they consider the parallel *substitution* (and not relation) $R1 \mid\mid R2$! This leads to the following error message :

```
Checking operation operation
Error: Invalid substitution R2
No information saved for Parallel1
End of Type checking
```

This error is obviously explained by the deep syntax ambiguity which cannot be removed without taking into consideration contextual information.

MACHINE
    $Parallel1$
VARIABLES
    R1, R2
INVARIANT
    $R1 \in \mathbb{N} <-> \mathbb{N} \land$
    $R2 \in \mathbb{N} <-> \mathbb{N}$
INITIALISATION
    $R1, R2 := \varnothing, \varnothing$
OPERATIONS
    $xx \longleftarrow operation =$
    BEGIN
    $xx := R1 \mid\mid R2$
    END
END

Fig. 1. Confusion with $||$

Note that one way to remove the confusion is to put parenthesis around the relational expression since parenthesis are allowed around expressions but not around substitutions. One may consider this hack as unnecessary and ugly.

Our tools [20] are able to parse correctly such sentences with the help of a few heuristics after lexical stage and before parsing stage, however, we have to say that this is not a clean language processing.

Others analoguous conflicts are enumerated in the figure 2.

Conclusion on operator overloading As we just have seen it, out of their context, some B sentences may have several meanings mainly because a few operators are overloaded. About such confusions [15, ch.2,p.16] wrote : *In precise formal specification, we cannot afford the confusion and should use distinct symbols for distinct operators.*

Separately taken, these problems seem rather anecdoctical but we have to keep in mind the essential difference beetween human and computer (tools) reading : overloadings are not genuine problems for human readers (as shown in example of figure 1) because human reader is able to perform the corresponding "contextual analysis" (provided that the formal text is not too long). Meanwhile, contextual analysis (implied by language processing) are not very easy for computer tools (if we restrict us to the well known language processing theory [4]).

In order to give a clean (syntactic) structure to the B language, we believe that is is necessary to remove the ambiguities we described in the last paragraphs and others which are enumerated in the figure 2.

Operator	Different Uses	(sub) Language
« ; »	Substitutions sequencing	Substitutions
	Relations composition	Expressions
	List separator	Clauses ...
« = »	Equality test in conditions	Predicates
	Set definition	Expressions
	Valuations	
	Beginning of operation bodies	Clauses
« : »	Membership	Predicates
	Become such as	Substitutions
	Record's field valuation	Expressions
« || »	Parallel composition for substitutions	Substitutions
	Parallel composition for relations	Expressions
« - »	Arithmetic difference	Expressions
	Sets difference	Expressions
	Unary minus operator	Expressions

Fig. 2. Overloading of operators in the *B* language

2 Proposed changes to the *B* language

In the field of software critical development, the *B methodology* will articulated around handling of properties known as *safety properties* which concentrate the essential requirements that the software must ensure in term of safety. To facilitate the analysis of these property, tools supporting the B method must allow the location, the management of their traceability compared to the external documentation and finally to follow their coherence when they be interdependent.

We evoked what would be a "property based" specification style by the mean of "cascaded" definitions. This organization makes it possible to define **explicit** links between the various properties and thus to limit the difficulties of comprehension and maintenance of the formal text.

The maximum benefit generated by the use of such a methodology could be reached only if the tools will be able to preserve these dependences. However, the expansion of the definitions has as a simple consequence, the loss of the original text structure and the loss of dependences information between definitions i.e. properties.

Thus, we consider that it is essential to put the B specification processing in a context where the tools are able to give a "clean existence" to the definitions. This clean existence implies at least the syntactic existence, i.e. the integration of the definitions clause in the grammar of the *B* language.

Our goal is now to show that it is possible to define a (BNF)[3] grammar for a *B* language without unnecessary syntactic conflicts and without mandatory

[3] Backus Naur Form[4]

DEFINITIONS clause expansion. The corresponding language may be slightly different from B and we call it B'.

2.1 Syntaxic integration of DEFINITIONS

This process of expansion is sketeched int the figure 3. The left part of the figure 3 presents the current process of processing of the definitions during the acquisition of a B component.

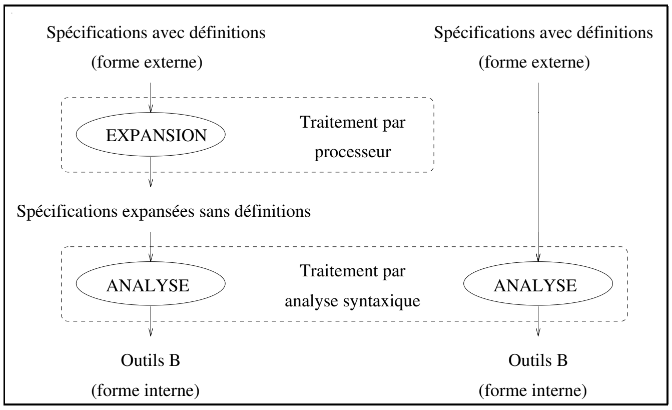

Fig. 3. Definitions (pre-)processing

The essential point lies in the fact that the syntactic analysis cannot be carried out *only after* the expansion of the definition's text.

Thus, the analyses cannot be performed on the "original" form (figure 4 first box) but only to the corresponding "expanded" (figure 4 second box).

On the other hand, the right part of the same figure proposes a simplified process where the textual expansion is not necessary any more and where the processing are thus carried out on the text of origin.

Currently, systematic expansion of the definitions prohibits to us to directly handle the original text of the specifications. To avoid this parasitic preprocessing, we must give to the DEFINITIONS clause a correct syntactic framework (satisfying the BNF standard). This framework is summarized with the statement of syntactic rules defining all the textual forms acceptable for the DEFINITIONS clause in a component B .

The usual documents of reference on B do not carry out this "syntactic integration". We give below the various rules given up to now for the syntax of clause DEFINITIONS in these reference documents. We indicate simultaneously the corresponding gaps that we have to fill.

```
MACHINE
    defs /* before expansion */
DEFINITIONS
    def1 == ..ConstX
    ; vars == var1, var2, var3
    ; prop1 == (p11 ∧ p12 ∧ ... ∧ p1n)
    ; prop2 == (p21 ∧ p22 ∧ ... ∧ p2n)
    ; prop3 == (prop1 ∧ prop2)
VARIABLES
    vars
INVARIANT
    vars ∈ prop3
END
```

```
MACHINE
    defs /* after expansion */
VARIABLES
    var1,var2,var3
INVARIANT
    var1, var2, var3 ∈ p11 ∧ p12 ∧ ... ∧ p1n
                           ∧ p21 ∧ p22 ∧ ... ∧ p2n
END
```

Fig. 4. DEFINITIONS clause expansion : an example.

2.2 Available grammars for the *B* language

Several grammars have been proposed:

- [1] contains a grammar which is more functional than effective, i.e is not very detailed and thus cannot be used as a starting point for the construction of tools like *B* parsers.

  $$Identifier(IdList) == FormalText$$

  However, no production rule is defined for *FormalText* !
- [8] was published during April 1995 by B-Core. The paper is largely available (on Internet) but is rather obsolete.
  This document do not provide syntactic rules for the DEFINITIONS clause because it consider that definitions are not syntactic entities. In other words, the proposed grammar only covers specifications without definitions.
- [19] is the more complete and uptodate reference manual for the *B* language. But, it still remains unsatisfactory because there is *no direct link* (like automatic generation) beetween the grammar specified in it and the *B* parser used in the corresponding tool. And last, the problems we have described in this paper are not solved.

  $$Definition ::= ident[(ident^{+,})] == lexeme^{*}$$

  Obviously, the preceding production rule is not correct because the non-terminal *lexeme* is not valid for the considered *BNF* grammar (and for any existing grammar because it is not a valid BNF construct).

We thus propose to supplement grammar using the rules *BNF* listed in figure 5 for clause DEFINITIONS . These rules allow the integration of the DEFINITIONS clause **in** the grammar (not done in [8] or [19]) and they give a formal definition to the syntactic entity *FormalText* used but non defined in [1].

Definition	::=	DefName "==" DefBody
DefBody	::=	Predicate
		Substitution
		Expr

Fig. 5. Syntactic integration of definitions

The integration of the definitions to the centres of grammar B as a syntactic object would allow to define rules of visibilities allowing, for example, the transitivity of the DEFINITIONS through clauses of connection.

This mechanism would be in conformity with the philosophy of project management of method B than the solution in the tools B , which consists in introducing the concept of file within the machines.

2.3 Removing operator overloadings

In the following, we propose some changes to the B language. These changes aim at removing a few operators overloading which are responsible of (too) many conflicts in the B grammar.

For now, these conflicts are resolved by the mean of heuristics during lexical analysis. These heuristics have not been published (as far as we know) and thus remain for validation.

These conflicts may arise when the concerned operators are involved in definition's bodies. Since we cannot suppress the DEFINITIONS clause, we have to change the notation used for some operators. We describe now a few examples of those changes.

Sequential composition Conflicts generated by the symbol $\boxed{;}$ are too significant, we have to reduce them. A modification of notation must not involve some disturbance significant for the notation and thus for the reading of text, it seems reasonable to modify the symbol of composition of relation. A simple solution consists in doubling the symbol.

We propose to change the sequential composition of relations from $\boxed{;}$ to $\boxed{;\,;}$.

Parallel composition The preceding conflict related to the use of a same symbol as operator on one hand and punctuation on the other hand. Here, the symbol $\boxed{||}$ is used for two different operators. This type of ambiguity prevents the use of the operator in a definition since one cannot make a syntactic analysis without having "contextual" information about the futur use (call) of the definition.

This kind of conflict have to be avoided, so we propose to incline the symbol to denote the parallel composition of relations. This modification is sufficiently tiny not to disturb the current users. We propose to change the sequential composition of relations from $\boxed{||}$ to $\boxed{//}$.

Set difference Integers differences and set differences are not syntactically different i.e they have the same syntactic form $A - B$. This nondesirable and surprising fact leads to a complex type checking process. however, we note that there is a specific and unambiguous notation for set differences, $A\ B$, and further more, this notation is used in *VDM* [4] and *Z* [5]. We simply suggest to change the set differences syntax from $A - B$ to $A \backslash B$.

Symbole	use	changed to
« ; »	substitutions sequencing relational composition list separator (operations, sets...)	 « ;; »
« : »	membership become such as record field valuation	 « = »
« = »	equality test between expressions (condition) set definition valuation beginning of operation body	
« || »	parallel composition of substitutions parallel composition of relations	 « // »
« - »	difference between arithmetic expressions unary minus operator set difference	 « \ »

Fig. 6. Proposed chages to remove (some) operators overloading

Figure 6 recapitulate the modification that we suggest for the notation *B*. This table take again the overload of operator that we have index and propose possibly in column of right-hand side a new notation for one of use of operator in cause.

3 Enhancing *B* redability

In order to obtain a *B* formalism which would lend itself better to the analyses, we can suggest other major modifications.

[4] `http://www.ifad.dk/vdm/bnf.html`

3.1 Separating invariant and typing

The first example of modification consists of merging the declaration of variables in clause VARIABLES and their corresponding typing in clause INVARIANT . This separate writing present, from our point of view, two disadvantages. First is that the current separation of two intrinsically dependent elements makes the reading of the specifications more difficult, in particular in the case of the significant projects where the number of variables strongly increases.

The second is the concentration the INVARIANT clause of a great number of expressions of different kind (typing of the variables, significant properties ...)

A more traditional writing, allowing us to declare and type a variable in a single expression, would be desirable. This must make it possible to reserve the invariant of component B for some property more critical (which have to be preserved by proof) that the simple coherence of type (preserved by the type checking).

VARIABLES N1, N2, N3 INVARIANT $N1 \in \mathbb{N} \wedge$ $N2 \in \mathbb{N} \wedge$ $N1 > N2 \wedge$ $N3 \in \mathbb{N} \wedge$ $(N1 + N2) > N3$	VARIABLES $N1 \in \mathbb{N} \wedge$ $N2 \in \mathbb{N} \wedge$ $N3 \in \mathbb{N}$ INVARIANT $N1 > N2 \wedge$ $(N1 + N2) > N3$

Fig. 7. Separating invariant from typing

A "typed" VARIABLES clause does make sens if the variables are used in distinct clauses as in the extensions of B where properties of behavior [2] are expressed, [3,11] in at least a new clause.

3.2 Typed interface

If we extend the preceding considerations to other entities of the B language like the operations, we can then highlight the concept of "typed interface".

Currently, the profile of a component B is given by the list of the names of its aggregate variables, the list of the names of the operations and parameters associated without any information about typing. The information which constitutes this profile is thus relatively poor.

To allow to work out an typed interface (or profile), we propose to clarify the typing of the parameters of the operations present in a component. We pass from the form (table 8-left) to the forme (table 8-right), much more «informative»

```
q ⟵ quotient(p, d) =
    PRE
        p ∈ ℕ ∧ d ∈ ℕ₁
    THEN ...
END
```

```
q ∈ ℕ ⟵ quotient(p ∈ ℕ, d ∈ ℕ₁) =
    BEGIN ...
    END
```

Fig. 8. Profile without types -- Profile with types

This richness makes it possible to engage certain processing or analyses without needing to examine the remainder of the operation (principle of separate compilation). This explicit typing could also be extended to other structures of the language:

```
VAR xx ∈ ℕ IN
    WHILE xx ≠ 10
DO ... END
```

```
VAR xx IN
    WHILE xx ≠ 10
DO ... END
```

Fig. 9. Profile without types -- Profile with types

The changes that we have just proposed pursue the same goal, opening the B language to external analyses. In fact, the expressive power of a formal language like B make it necessary to «unfold» the formal text in order to make it available to the more traditional techniques. In short, for a better integration (opening) of formal methods, it is significant not to be let lock up by a too thorough formalism.

These modifications bring other great benefits such as better a readability, a clear interface and the definition of a more explicit typing.

4 Conclusion

The broader adoption of the formal techniques by the industrial world depends on a better integration in the process of software development.

Following this point of vue, [7] describes a structure designed to software formal development by facilitating information circulation beetween different involved tools or actors. We may speak about «Integrated Formal Development Environment (IFDE)» and no more about «Integrated Development Environment (IDE)»

Figure 10 is a simplistic view of the choosen structure. As we can see, IFDEs should include tools for getting (file systems, editors), processing (proof system, model checking, animators, tests suite), and displaying (documentation, translators). Every element of this environment may speak with others by the mean of a central repository. This repository stores every information which may be needed by one particular tool. The notation used used at this place have to be generic across the tools. The essential question is the choice of such notation.

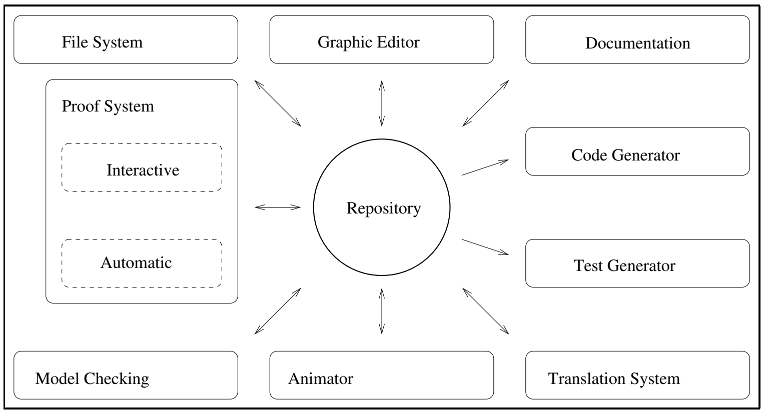

Fig. 10. Integrated Formal Development Environment

An open structure is a guarantee for stable systems and thus, it is essential to define a common notation in particular when the main goal is the spread of the formal method across software development process (from specifcatoin to code).

As a first step, we use the syntactic description of the B formal text. This description is easily obtained when a "BNF" grammar is available.

The existence of a grammar *BNF* for a given language is a stake not only *scientific*, the language has " good " properties, but also a stake *technologic* because this grammar facilitates the rigorous development of any tool for handling the language structures (analyses, interpretation, compilation, ...

Moreover, the notation *BNF* is the only formalism of description of syntaxes of languages currently standardized. This is the fisrt step for a standardization of the language B , this one being a strong argument in favour of the diffusion of the language on an international scale.

As an example, we can note the broad diffusion of grammars *BNF* for other formal methods, in particular *Z* and *VDM* which belong to the same family tahn B . Another example, among othesr, is the language *VERDI* supporting the formalism *EVES* ([17]). For each one of these languages, the broad availability of these documents is a pledge of opening.

4.1 Futur work

Static analysis on B formal specification (measurement, testing, automatic documentation) have to be based on the availability of a B language grammar. This grammar is an abstract model for formal texts on which we will define systematic analysis.

We have reach this first goal by constructing a working gammar following well known technics (lexer and parser construction).

Now we may also consider that this grammar is only the first stage because BNF grammar only deals with «concrete (textual) form», that is from an external view [15, p.?].

Syntactic analysises will not reach more sophisticated concepts like type checking, modularity (entities sharing), proofs... These concepts need a more abstract model closer to semantic considerations.

One way of investigation is the use of attributed grammar (which can be viewed, in a first sight, as decorated syntactic trees). We are currently exploring this path with the help of automatic tools called SYNTAX/FNC2[10] developped at INRIA[5] during OSCAR project [6].

These tools allow the **specification** and the automatic generation of common tools such as type checkers, documentation generators, translators...

A second study (following the building of the BNF grammar) will focus on building an attributed grammar (which is also an executable specificatoin) for B type checking.

Indeed, some aspects of B type checking remain unclear as shown in the following example. Despite its simplistic apparence, the following operation cannot be typed by available tools [7].

OPERATIONS $xx \longleftarrow op$ = BEGIN $xx := \varnothing$ END

```
Error: Parameter xx has not been typed
```

The next step will be the specification, based on attributed grammars, of the B type checking system (i.e types considered as attributes). Validation of the resulting specification will be done by generating a type calculator tool.

In the futur, this work will lead us to formalize, in the same framework, aspects related with automatic code production, that is automatic translation from B specifications to various programming languages (C, ADA, Java...).

References

1. Jean-Raymond Abrial. *The B Book - Assigning Programs to Meanings.* Cambridge University Press, August 1996.
2. Jean-Raymond Abrial. Extending B without changing it (for developing distributed systems). In Henri Habrias, editor, *Proceedingsof 1st Conference on the B method*,

[5] Computing and Automatic National Research Institute
[6] Syntactic tOols for pRogram Construction and Analysis
[7] Note that with explicit typing as exposed in 3.2, this is not a problem any more because we have to type the output parameter xx

Putting into Practice methods and tools for information system design, pages 169–191, 3 rue du Maréchal Joffre, BP 34103, 44041 Nantes Cedex 1, November 1996. IRIN Institut de recherche en informatique de Nantes.
3. Jean-Raymond Abrial and Louis Mussat. Specification and design of a transmission protocol by successive refinements using b. In Manfred Broy and Birgit Schieder, editors, *Mathematical Methods in Program Development*, volume 158 of *NATO ASI Series F: Computer ans Systems Sciences*, pages 129–200. Springer, 1997.
4. Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilateurs, principes, techniques et outils.* Informatique Intelligence Artificielle. InterÉditions, 1989.
5. Rodger Collinson. The matehmatical symbols of Z and VDM-SL: A comparison. Z Standard Process, feb 1995.
6. Mariano Georges. *Évaluation de logiciels critiques développés par la méthode B : une approche quantitative.* PhD thesis, Universitée de Valenciennes et du Hainaut-Cambrésis, Dec 1997.
7. A. M. Gravell and C. H. Pratten. Formal methods and open systems. *Software—Concepts and Tools*, 16(4):183–188, 1995.
8. Dick Jeremy. AMN notation. Technical report, BCore Ltd, April 1995.
9. Cliff B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall International, Englewood Cliffs, New Jersey, second edition, 1990. ISBN 0-13-880733-7.
10. Martin Jourdan and Didier Parigot. Application development with the FNC-2 attribute grammar system. In Dieter Hammer, editor, *Compiler Compilers '90*, Lecture Notes in Computer Science, pages 85–94. Springer-Verlag, New York–Heidelberg–Berlin, October 1990 January 1992. Schwerin.
11. J. Julliand, B. Legeard, T. Machicoane, B. Parreaux, and B. Tatibouet. Specification of an integrated circuit card. protocol application using the B method and linear temporal logic. In Didier Bert, editor, *B'98 : The 2nd International B Conference*, Lecture Notes in Computer Science 1393, pages 273–292, Montpellier, Apr 1998. LIRRM Laboratoire d'Informatique, de Robotique et de Micro-électronique de Montpellier, Springer Verlag.
12. Kevin Lano. Formal development in B Abstract Machine Notation : FME'96 Tutorial, 1996.
13. Georges Mariano. Une grammaire BNF pour le formalisme B - v0.3. Technical Report 96-24, INRETS–ESTAS, Septembre 1996.
14. Georges Mariano and ElMiloudi ElKoursi. Formal methods and metrics: Application to B development assessment. In W.B. Samson, I.M. Marshall, and D.G. Edgar-Nevill, editors, *Proceedings of the 5th Software Quality Conference*, pages 37–55, University of Abertay Dundee, Bell Street, Dundee DD1 HG, July 1996.
15. Bertrand Meyer. *Introduction to the theory of programming languages.* Prentice Hall, 1991.
16. C. Morgan. *Programming from Specifications.* International Series in Computer Science. Prentice Hall, 1990.
17. Mark Saaltink. The lexical structure of VERDI. Technical Report TR-94-5463-06, ORA Canada, July 1994.
18. J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall International Series in Computer Science, 2nd edition, 1992.
19. Stéria. Le langage B : Manuel de référence. Technical Report V 1.8, RATP SNCF INRETS, June 1998.
20. Bruno Tatibouët. A grammar and a parser for B. Technical Report will be soon available, Laboratoire d'Informatique – Besançon, feb 1999.

List of Figures