A Study on
Components and Assembly Primitives in B

Didier Bert, Marie-Laure Potet, Yann Rouzaud
LSR-IMAG, Grenoble, France

Laboratoire Logiciels Systèmes Réseaux - Institut IMAG (UJF - INPG - CNRS)
BP 53, F-38041 Grenoble Cedex 9 - Tel + 33 4 76827214 - Fax + 33 4 76827287
e-mail: Dider.Bert@Imag.fr, Marie-Laure.Potet@imag.fr, Yann.Rouzaud@imag.fr

Abstract. This paper is the result of a reflexion coming from the usage and learning of the language B. It tries to better explain and understand the assembly primitives INCLUDES and USES of the language. It presents a high-level notion of components and develops a "component algebra". This algebra is specialized to deal with the B-components. The B assembly primitives are re-expressed in this basic formalism. Some problems about independence of concepts in the B methodology are pointed out and are discussed.

1 Introduction

Specifications, like programs, must be modular because very large formal texts are not understandable for a human being. So, the study of modules and modularization is one of the issues in software engineering. The three main objectives of modularization [BHK90] are : information hiding, compositionality of module operations and reusability of modules. If the specification methodology encompasses the need for formal proofs to ensure consistency, as it is the case in the B method, then modularity is also very crucial to decompose the proof process in many little steps, which are much more tractable than one very large proof. However, even if a specification language is "modular", it must be clear what is the basis of such a concept (not uniquely a syntactic facility), what is its semantic meaning and, consequently, what is its impact on the specification design.

An answer to these questions is to try to define a notion of *component*, as much as possible independent of any particular language and to have composition operations on the components at a semantic level. Then we can apply or instantiate the proposed model of components to a given specification language.

In this paper, we are interested in the notion of modularity in the B methodology. By some aspects, abstract machines can be considered as usual modules because they contain an internal state and export operations on this state. Interfaces of *implemented* abstract machines are layers and provide "services". A layer is opaque because one cannot (or does not want to) know how the machine runs, e.g. if it is a software or a hardware machine, and so on. But, at the level of specification, i. e. machine construction, the notion of machine is not exactly the same. An analysis of the B language definition shows quickly that abstract machines are also pieces of specifications which can be combined in several ways and must be refined, all things very different from usual programming modules. Even a comparison with the object methodology fails with respect to several points, mainly on the point of the modular decomposition of an application. So, a user can be bothered when (s)he has to design an application with the B method, or when (s)he wants to define reusable abstract machines for further use.

From a decomposition point of view, there are three main aspects of structuring in the B methodology. They are :

1. Using ou reusing some machines (considered as pieces of specifications) to build new larger machines. The aim is to gather texts of machine specifications. Once the new machine is built, the initial submodules are no longer useful because they are "copied" in some sense in the new machine. Clauses involved for this assembly are : INCLUDES and USES. They can be considered as *weak relations* because they can be forgotten when considering the complete software.
2. Drawing a refinement development. This view concerns an initial abstract machine and the list of refinements of this machine until the implementation. Such a *development* must contain the decisions about the choices done in each refinement step. This is the view of the development (and coding) of

a machine. It is reflected by the primitive REFINES. It can be useful for the reviews. It is the main document for a safe maintenance activity and to replay developments in case of evolution of the requirements.

3. Building a software architecture of modules. In this case, structuring clauses correspond to *strong relations* because they are always visible in the final architecture. They connect machines which remain as software components of the application. These clauses are, of course, SEES and IMPORTS.

One aim of this paper is to try to better understand the *assembly principles* of abstract machines. So, we address only the first point quoted just above. However, we do not intend to explain what is the "philosophy" of the B method or to give recipes for building sensible abstract machines. Rather, we want to study what are the composition clauses of the B language and to formalize some parts of the assembly mechanisms. A key point is to study assembly clauses with respect to several properties like orthogonality (i.e. independence of constructions), reusability of parts of design, reusability of proofs, etc. Our approach starts with the definition of a high-level notion of component. Basic operations to combine components are presented. Then, description of the components and assembly clauses of the B language is done in terms of this general approach.

In the first part (section 2), the notion of components is introduced and a set of composition operations on components (i.e. assembly primitives) is described. This framework is general enough to cover features of many modular specification languages. In section 3, the B abstract machines are presented together with the assembly clauses INCLUDES and USES. Some specific features of the abstract machine notation, like the rules of well-definedness of the machines are explained. The main differences between both assembly clauses are given. We are reminded of the constraints associated to these clauses and a discussion about their methodological impact on machine construction is initiated. Then in section 4, our notion of components is applied to the B language. The operations on components are specialized for B and verification conditions are detailed in each case. The rest of the paper (section 5) is devoted to the (re)definition of the assembly clauses INCLUDES and USES in term of our basic primitives. Then, in the conclusion, we expose some remarks about the actual definition of the assembly clauses in B.

2 Software Components and Assembly Operations

2.1 What is a component ?

To define the word "component", we must start with the notion of *modular language*. In a modular language, a module is a text which enjoys several properties. These properties are generally: separate parsing, separate compiling, information hiding, visibility rules to export a part of the content (notion of interface). The language provides assembly primitives to relate or to connect modules one each others. Modules used in another module can be called *submodules*. A *component* is defined in this paper as an entity associated with a module, if some well-definedness rules are satisfied. This entity has a meaning by itself. This meaning is obtained from the content of a module and from the interpretation of the assembly primitives which are written in the module. Sometimes, the component of a module M is defined as a *flattened module*, i.e. a module containing the text of the (inductively) flattened submodules concatenated with the text of the own declarations of M. In that case, a component is a basic large module. In other approaches [SJ95] [Ori96], modules are seen as points in a categorical framework, where the assembly primitives are interpreted as morphisms, and a component is defined as the *colimit* of such a diagram of specifications. At last, the notion of components can be entirely a semantical notion (without concrete representation) like in [BG77], where the meaning of an assembly of modules is a theory and in [Wir86] [ST88], where the meaning of a structured algebraic specification is a class of algebras. Several views of components may coexist and can be related. This has been formalized between theories and classes of algebras (models) in the framework of the Institutions [GB90]. Other attempts connect flattened specifications and algebras [EFH83], but for complex structuring primitives, the correspondence between several semantical views of modules can only be partial [Fey88] [BE95].

Because a component is an abstract notion, we can define operations on components in a mathematical way (component algebra). Assembly clauses on modules (e.g. INCLUDES) must have a semantic interpretation as operations on components. So, a language is more or less "modular" according to the more or less strong properties of its components. For instance, the semantics of a language is a modular semantics if the semantics of an assembly of modules is defined by a semantical operation applied to the components associated with the submodules.

Very generally, a component C is characterized by a set of identifiers which are declared and constitute the visible *signature* of the component. These identifiers are usually typed and can be functions, operations, variables, constants, etc. The notion of "type" in this introduction is left vague and should be made precise for each particular language. The visible signature of a component C is a set of identifiers (with their type) denoted by $vis(C)$. Visible identifiers are divided into parameters: $par(C)$ and declared identifiers: $dec(C)$. In a component, a part of the information can be hidden, so it is sometimes needed to deal with a hidden signature, called $hid(C)$.

The second main part of a component is the *body*. Identifiers are specified in the body of the components either by their meaning or their properties (e.g. operations), or by their range or invariant (e.g. variables), or by their value (e.g. constants) or by something else (again, this definition remains intentionally fuzzy). Parameters are specified, but they have no specific value. Identifiers and their specification constitute the local environment of a component. Some conditions (or constraints) can be explicitely stated on this environment like assertions, invariant, etc. All the information associated with the identifiers is called the body of the component. So, we shall use the expression "a Σ-body B" to speak of an environment B on the identifiers Σ. Conditions of well-definedness of a component can be formalized in this notation. They are validity or consistency conditions depending on the signature Σ.

To summarize, in a component C, there are a signature $sig(C)$, a body $bod(C)$ and consistency conditions $coc(C)$. The set of the identifiers known in C satisfy:

$$sig(C) = vis(C) \cup hid(C) \text{ with } vis(C) \cap hid(C) = \emptyset,$$
$$\text{and } vis(C) = par(C) \cup dec(C) \text{ with } par(C) \cap dec(C) = \emptyset.$$

A component will be denoted by a pair (Σ, B) or to be more precise by a 4-tuple:

$$C = (P, V, H, B)$$

with $P = par(C)$, $V = dec(C)$, $H = hid(C)$ and B is the $(P \cup V \cup H)$-body of C satisfying $coc(P, V, H, B)$.

The consistency conditions of C are not an element of the definition of the components, because we do not want to include the semantical conditions (proof obligations) and their proofs in the language of components. For example, this has been done for algebraic specifications in [BL91].

2.2 Auxiliary operations

Operations on signatures. Signatures essentially are sets of names. So the operations on sets (union, intersection, set difference, ...) can be applied to signatures or to signature parts (visible, hidden, etc.). Let Σ_1 be (P_1, V_1, H_1) and Σ_2 be (P_2, V_2, H_2), then $\Sigma_1 \cup \Sigma_2$ denotes the pointwise union of signature parts $(P_1 \cup P_2, V_1 \cup V_2, H_1 \cup H_2)$.

Moreover, given two signatures Σ_1, Σ_2, a *signature mapping* $\sigma \in \Sigma_1 \longrightarrow \Sigma_2$ also denoted $\Sigma_1 \xrightarrow{\sigma} \Sigma_2$ is a correspondence between the identifiers of Σ_1 and those of Σ_2 which is compatible with their type. If the mapping is injective, (two different identifiers are not confused in the target signature) then it is called a *renaming*. The identity signature mapping is denoted by $id(\Sigma)$ (names of the source signature Σ are not changed).

Operations on bodies. Because the notion of "body" is not completely defined, then the operations given here are only intuitively presented. In some sense, they are formal operations which must be available to combine components.

Let B_1 be a Σ_1-body and B_2 be a Σ_2-body, then we denote by $B_1 \otimes B_2$ the *merge* of the bodies. The result is a body on $\Sigma_1 \cup \Sigma_2$, where the specifications coming from both bodies on the same objects are "cumulated". In the case where Σ_1 and Σ_2 are disjoint, then the operation merge is equivalent to the *union* of the two bodies, denoted $B_1 \oplus B_2$. This union can be considered as the concatenation of the two bodies because they do not share any identifiers.

Again, let B_1 be a Σ_1-body and $\Sigma_1 \subseteq \Sigma_2$. Assume that B_2 is a set of body information on Σ_2, then $B_1 \rhd B_2$ is a Σ_2-body where B_1 is *enriched* by B_2. The operation $\rhd$ on the bodies means that the right part is added to the body given at the left part. A single body can be considered as an enrichment of the empty $\emptyset$-body.

At last, let σ be a signature mapping $\Sigma_1 \xrightarrow{\sigma} \Sigma_2$ and B be a Σ_1-body, then $\sigma(B)$ is a Σ_2-body obtained by the application of the mapping σ to the body B. The result of the application is that each occurrence of a name n in B such that $n \in dom(\sigma)$ is replaced by its value $\sigma(n)$.

2.3 Operations on Components

To define a set of operations on components, at this level of abstraction, we have taken into account three things: the information available in a component (signature, body, etc.), what we want to do with this information (e.g. renaming the identifiers) and what are the primitives usually found in specification languages. So, many of the operations proposed below exist in modular languages under various forms. We choose to define here very primitive operations, in order to keep their meaning clear, although rather informal. All these operations are defined if the resulting component is "valid", i.e. satisfy the consistency conditions required by the language semantics. In case of B-components these consistency conditions will be explicitely stated in section 4.

Promotion. Promotion consists in making some hidden identifiers visible in the signature of a component. Let C be a component (P, V, H, B) and E be a set of identifiers such that $E \subseteq H$, then:

$$\textbf{promote } E \textbf{ in } C \ = (P, V \cup E, H - E, B)$$

Hiding. Hiding is the operation symetric to the promotion. It consists in making some visible identifiers hidden in the signature of a component. Let C be a component (P, V, H, B) and E be a set of identifiers such that $E \subseteq V$, then:

$$\textbf{hide } E \textbf{ in } C \ = (P, V - E, H \cup E, B)$$

Instantiation of parameters. A parametrized component can be instantiated, that is to say, parameters are given a value (actual parameter) compatible with their type. The instantiation process can be expressed as a substitution of actual values for formal names. Let C be a component (P, V, H, B) and let σ be a substitution of the form $\{x_i \mapsto v_i\}$ for $i \in [1..k]$, where $dom(\sigma) \subseteq P$, and for each association $(x \mapsto v)$, v is "compatible" with the type of x and v may only contain free variables of V, then:

$$\textbf{instantiate } C \textbf{ by } \sigma \ = \ (P - dom(\sigma), V, H, \sigma(B))$$

This definition allows us to take into account partial instantiation of the parameters of a component.

Renaming. Renaming consists in changing consistently the name of some identifiers in the signature. It is expressible by a substitution of names. A renaming $\Sigma_1 \xrightarrow{\sigma} \Sigma_2$, where $\Sigma_1 = (P_1, V_1, H_1)$ and $\Sigma_2 = (P_2, V_2, H_2)$ is the union of the three renamings $P_1 \xrightarrow{\sigma_p} P_2$, $V_1 \xrightarrow{\sigma_v} V_2$, $H_1 \xrightarrow{\sigma_h} H_2$. This union is well defined because the source sets are disjoint. It is needed that the mode (parameter, declared or hidden) is preserved in the target signature. Given a component $C = (\Sigma_1, B)$ and a renaming $\Sigma_1 \xrightarrow{\sigma} \Sigma_2$, then:

$$\textbf{rename } C \textbf{ by } \sigma \ = (\Sigma_2, \sigma(B))$$

Union with implicit sharing. The union consists in building a new component by putting together the informations coming from two given components C_1 and C_2. If the two component have two (or more) names identical and if these identifiers represent the same entity in such a way that only one occurrence of this entity is expected in the resulting component, then we have to do a union with sharing. In that case, we can do a set theoretic union of the identifiers and this operation "identifies" the identifiers which are equal. However, specification and constraints on these identifiers coming from both components C_1 and C_2 are cumulated and must be compatible to ensure the validity of the resulting component. Moreover, the shared identifiers must belong respectively to the same part of the components (parameters, declared or hidden). Let C_1 be (Σ_1, B_1) and let C_2 be (Σ_2, B_2), with $\Sigma_1 = (P_1, V_1, H_1)$, $\Sigma_2 = (P_2, V_2, H_2)$ such that $P_1 \cap (V_2 \cup H_2) = \emptyset$, $V_1 \cap (P_2 \cup H_2) = \emptyset$, $H_1 \cap (P_2 \cup V_2) = \emptyset$ and symetrically for the identifiers of C_2 then:

$$C_1 \otimes C_2 \;= (\Sigma_1 \cup \Sigma_2, B_1 \otimes B_2)$$

Now, assume that we want to take the union of two components where there are some names identical in both components but not representing the same entity, from the point of view of the resulting machine. In that case, it is sufficient to rename one machine with fresh identifiers, and then to do the union operation. In the resulting machine, the identifiers will denote distinct entities. This method can be used to duplicate a component, for example.

If the signatures of the two components are disjoint, then the union with sharing becomes a simple union of the components noted $\oplus$. In that case, we can use the union operation on bodies:

$$\Sigma_1 \cap \Sigma_2 \;= \emptyset \Rightarrow \quad C_1 \otimes C_2 \;=\; C_1 \oplus C_2 = \;(\Sigma_1 \cup \Sigma_2, B_1 \oplus B_2)$$

Enrichment. The enrichment consists in adding new information to an "old" component. In terms of our notations, this can be expressed as follows. Given a component $C = (\Sigma_1, B_1)$ and $\Delta C = (\Sigma_2, B_2)$ with $\Sigma_1 = (P_1, V_1, H_1)$ and $\Sigma_2 = (P_2, V_2, H_2)$, $\Sigma_1 \cap \Sigma_2 = \emptyset$ and B_2 is a $(P_1 \cup P_2, V_1 \cup V_2, H_2)$-body (or elements of such a body) then:

$$\textbf{enrich } C \textbf{ by } \Delta C \;= (\Sigma_1 \cup \Sigma_2, B_1 \rhd B_2)$$

2.4 Component Algebra

Definition of basic operations on components is very useful for reasoning about assemblies of components. A composition of components can be written as a *term* which represents the structure of a construction. Some laws can be stated one for all and can be used to formally transform terms (i.e. specification expressions). A simple law example is the following rule:

$$\textbf{rename } (\textbf{rename } C \textbf{ by } \sigma_1) \textbf{ by } \sigma_2 \;=\; \textbf{rename } C \textbf{ by } \sigma_2 \circ \sigma_1$$

Many other rules could be given. The goal of the paper is not to develop such an algebra, so we do not seek to find out rule sets nor to discuss properties like minimality, soundness and completeness of these sets.

Operations on components have been defined in several specification languages. In *algebraic specifications*, a formalism like ASL [Wir86] provides operations on specifications which could be expressed in the framework presented in the section 2.3, at least if they are not too specific and do not depend on the algebraic semantics of the components. The primitives of the language are: enrich, rename, union with implicit sharing, instantiate, quotient (which can be represented by our enrich), etc. One can found a complete study of meta-operations on algebraic specifications called "modules" in [BHK90].

Model-oriented specification formalisms provide examples of primitives to assemble specifications. The *Z language* [Spi88] is modular, because all the schemas can be specified separately. Bodies of component are texts of specification and can be obtained by replacement and copy of texts of the subcomponents. Useful operations to describe the assembly primitives in Z are rename, union with implicit sharing, enrich, etc.

3 Assembly Clauses in B

3.1 Modules in B

In the language B, abstract machines can be considered as modules, as explained in section 2.1. The following figure shows a simple machine, with no assembly clauses. Some features have been simplified[1] with respect to the full definition of machines given in the B-book [Abr96].

[1] In this paper, we do not describe all the elements of the abstract machines. Intentionally, we drop the parts SETS, CONSTANTS, PROPERTIES, and the conditions associated to the parameters (non emptyness and finiteness of the sets, etc).

MACHINE	
$M(P)$	Name and parameters of the machine (list of identifiers).
CONSTRAINTS	
K	Definition of constraints on the parameters (predicate).
VARIABLES	
X	Definition of the variables describing the state (list of identifiers).
INVARIANT	
I	Definition of the invariant on X (predicate).
INITIALISATION	
U	Initialisation of the variables X (generalized substitution).
OPERATIONS	
O	Definition of operations, of the general form : $o =$ PRE Q THEN S END, where Q is a predicate and S is a generalized substitution. We note $dom(O)$ the set of operation names.
END	

An abstract machine is *consistent* if, and only if the following conditions (called "proof obligations") hold:

1- initialisation sets up the invariant: $K \Rightarrow [U]I$

2- operations preserve the invariant: $K \wedge I \wedge Q \Rightarrow [S]I$ for each operation $o \in dom(O)$

(to be more precise, initialisation must set up all the variables of the machine; this is a syntactic restriction which will be used in the next sections).

Some assembly clauses are offered in order to compose abstract machines, together with their proof obligations. These clauses can only appear inside abstract machines : they are not operations of module composition. At the level of abstract machines, the two assembly clauses USES and INCLUDES allow us to elaborate union of abstract machines, combined with enrichments: the clause INCLUDES creates local copies of instantiated abstract machines, whereas the clause USES prepares some sharing. For instance if we want to share a machine M_1 by two others machines M_2 and M_3 the construction will be:

$$\begin{tikzcd}
 & M_4 \arrow[dl, "\text{includes}"'] \arrow[dr, "\text{includes}"] \arrow[dd, "\text{includes}"] & \\
M_2 \arrow[dr, "\text{uses}"'] & & M_3 \arrow[dl, "\text{uses}"] \\
 & M_1 &
\end{tikzcd}$$

M_4 includes machines M_2 and M_3
which share the abstract machine M_1.
M_1 must be explicitly included in M_4 too.

The machine M_2 and M_3 extend, in some sense, the machine M_1 knowing that the machine M_1 will be shared. In the last step, machines using M_1 must be included together with M_1, in one machine M_4. We call this construction *a closure of a* USES *construction.* At the end of this construction, a new abstract machine with copies of M_1, M_2, M_3 is obtained, in which the local copy of M_1 is shared.

In the next sections, we briefly describe the effect of the clause INCLUDES when no clause USES appears in the included machines, then the effect of the clause USES and the treatement of the closure.

3.2 Description of the clause INCLUDES effect

A machine with a clause INCLUDES can easily be interpreted as an equivalent machine, without assembly clauses. Thus the consistency of this new construction can be stated in terms of the consistency of the resulting machine. For instance, consider the following construction where:

- I'_1 and U'_1 stand for formulae obtained from I_1 and U_1 after substitution of P_1 by E_1;
- o'_1 are definitions of the promoted operations o_1, after substitution of P_1 by E_1.
- O'_2 stands for O_2 after substitution of calls of the M_1 operations by their definition. This substitution is possible because no recursive definition is allowed in B. Generalized substitutions with calls can be always expanded in generalized substitutions without call.

```
MACHINE
    $M_1(P_1)$
CONSTRAINTS
    $K_1$
VARIABLES
    $X_1$
INVARIANT
    $I_1$
INITIALISATION
    $U_1$
OPERATIONS
    $O_1$
END
```

M_1 is included in M_2:

```
MACHINE
    $M_2(P_2)$
INCLUDES
    $M_1(E_1)$
PROMOTES
    $o_1$
/* operations of $M_1$ which
become operations of $M_2$ */
CONSTRAINTS
    $K_2$
VARIABLES
    $X_2$
INVARIANT
    $I_2$
INITIALISATION
    $U_2$
OPERATIONS
    $O_2$
END
```

The resulting abstract machine equivalent to M_2 is:

```
MACHINE
    $M_2(P_2)$
CONSTRAINTS
    $K_2$
VARIABLES
    $X_2, X_1$
INVARIANT
    $I_2 \wedge I'_1$
INITIALISATION
    $U'_1; U_2$
OPERATIONS
    $O'_2, o'_1$
END
```

In order to reuse proof obligations the clause INCLUDES also imposes some syntactic restrictions:

- signatures must be disjoint;
- the operations of the including machine do not directly modify the variables of the included machines;
- the calls of operations of the included machines must be controlled: in a $S_1||S_2$ substitution, S_1 and S_2 do not both contain a call of operations which modify some variables defined in a same included machine[2].

In the above construction the syntactic restrictions ensure that operations in $dom(O_2)$ preserve the invariant I_1, and *a fortiori* I'_1. Thus the local proofs are restricted to show that operations offered by the machine M_2, i.e. $dom(O_2) \cup o_1$, preserve I_2.

Example 1.

```
MACHINE
    $Ma_1$
VARIABLES
    $x, y$
INVARIANT
    $x \in \mathrm{NAT} \wedge \mathrm{y} \in \mathrm{NAT} \wedge \mathrm{x} < \mathrm{y}$
INITIALISATION
    $x, y := 0, 1$
OPERATIONS
    $op_1 = S_1; op_2 = S_2$
END
```

Ma_1 is included in Ma_2 :

```
MACHINE
    $Ma_2$
INCLUDES
    $Ma_1$
PROMOTES
    $op_1$
VARIABLES
    $z$
INVARIANT
    $z \in \mathrm{NAT} \wedge \mathrm{z} = \mathrm{x} + \mathrm{y}$
INITIALISATION
    $z := x + y$
OPERATIONS
    $op_3 = S_3$
END
```

In this example we indicate what are the validity constraints on each operation, but we do not want to detail the substitutions S_1, S_2 et S_3. In terms of consistency, at the end of this construction, we have:

[2] $S_1||S_2$ is well-defined if, and only if, the sets of variables modifyed by S_1 and S_2 are disjoint.

– op_1 and op_3 preserve the formula $x \in \mathrm{NAT} \wedge \mathrm{y} \in \mathrm{NAT} \wedge \mathrm{x} < \mathrm{y} \wedge \mathrm{z} \in \mathrm{NAT} \wedge \mathrm{z} = \mathrm{x} + \mathrm{y}$;
– op_2 preserves the formula $x \in \mathrm{NAT} \wedge \mathrm{y} \in \mathrm{NAT} \wedge \mathrm{x} < \mathrm{y}$.

3.3 Description of the clause USES effect

The construction USES is more complex. No interpretation, in terms of new abstract machines, can be given. This clause really takes a sense in the final closure of the sharing construction. Let M_1 and M_2 be the following abstract machines:

```
MACHINE
    M1(P1)
CONSTRAINTS
    K1
VARIABLES
    X1
INVARIANT
    I1
INITIALISATION
    U1
OPERATIONS
    O1
END
```

M_1 is used by M_2 :

```
MACHINE
    M2(P2)
USES
    M1
CONSTRAINTS
    K2
VARIABLES
    X2
INVARIANT
    I2
INITIALISATION
    U2
OPERATIONS
    O2
END
```

There is no resulting abstract machine equivalent to M_2.

Proofs obligations associated with the machine M_2 consist in ensuring that the operations defined in O_2 preserve the part of the invariant I_2 which is independent of variables X_1. Thus some proofs are delayed until the elaboration of the final closure if it turns out that they are needed: these proof obligations are relative to the preservation of the shared part of the final invariant for operations which will be promoted in the final closure. Some syntactic restrictions are also imposed by the method B:

– all signatures must be disjoint.
– the operations of the using machine can only read variables of the used machines.

The last point implies that operations of M_2 preserve the invariant I_1, variables X_1 being never modified. Moreover, because operations of machine M_2 do not modify in any way X_1, operations of M_2 and M_1 can be put in parallel in the final closure, as it is accepted in a clause INCLUDES for different machines.

Example 2.

```
MACHINE
    Ma3
VARIABLES
    z
USES
    Ma1
INVARIANT
    z ∈ NAT ∧ z = x + y
INITIALISATION
    z := x + y
OPERATIONS
    op3 = S3
END
```

Ma_1 and Ma_3 are included into Ma_4 :

```
MACHINE
    Ma4
INCLUDES
    Ma1, Ma3
PROMOTES
    op1, op3
END
```

In terms of consistency we have:

- in machine Ma_1, op_1 and op_2 preserve the formula $x \in \mathrm{NAT} \wedge \mathrm{y} \in \mathrm{NAT} \wedge \mathrm{x} < \mathrm{y}$.
- In machine Ma_3, op_3 preserves the formula $x \in \mathrm{NAT} \wedge \mathrm{y} \in \mathrm{NAT} \wedge \mathrm{x} < \mathrm{y} \wedge \mathrm{z} \in \mathrm{NAT}$. Locally, required proofs consist in showing that op_3 preserves $z \in \mathrm{NAT}$.
- In machine Ma_4, op_1 and op_3 preserve the formula $x \in \mathrm{NAT} \wedge \mathrm{y} \in \mathrm{NAT} \wedge \mathrm{x} < \mathrm{y} \wedge \mathrm{z} \in \mathrm{NAT} \wedge \mathrm{z} = \mathrm{x} + \mathrm{y}$. Locally, required proofs consist in showing that op_1 and op_3 preserve $z = x + y$.

Notice that, if op_3 is not promoted in Ma_4, it is never proved that op_3 preserves the formula $z = x + y$.

3.4 Assembly clauses are not component operations

A machine with a clause USES must be seen as an intermediary step of a structured construction, without real semantical content. From a methodological point of view some remarks can be stated:

- The text of a machine with a clause USES does not reflect a well-defined notion of consistency. Some parts of the invariant are not proved. From a methodological point of view, some intermediary constructions, which seem well-defined, can be called into question. For instance in Example 2, when op_3 is promoted, we have to prove that this operation preserves the second part of the invariant, $z = x + y$, asserted in the same machine in which it is defined.
- Some compositions of the USES and INCLUDES clauses are not allowed. For instance, machine Ma_3 cannot be included in another machine without Ma_1. We cannot delay the inclusion of an used machine in another level of inclusion.
- Proofs obligations of the INCLUDES construction are not managed in the same way, if it is a simple construction or a closure construction (proof obligations delayed).
- Because some proofs can be delayed, they can be proved several times if the intermediary construction is reused. In fact a clause USES is only a piece of text which cannot be seen as an independent specification.
- If we want to consider an architecture as a term, in order to reason about it, we have to deal with two levels of granularity: simple machines and constructions of sharing closure.

4 B Components

In the last part of this paper we define a notion of B-component in order to revisite the clauses USES and INCLUDES as component operations. A comparaison, from the proof obligations point of view and from the methodological point of view will be done.

4.1 Definition of B components

We give in the following figure the composition of each field of a B-component and the nature of the information. It is intended that an operator name is associated with an arity (its "type") and the arity contains the names of the parameters and results. So, these names do not appear as identifiers in a component, but are accessible in meta-operations that involve operator names.

Name of the field	Content	Comment
Parameters	P	List of the parameters names.
Visible names	$\langle V_x, V_o \rangle$	The meta-variables V_x and H_x mean B-variable names
Hidden names	$\langle H_x, H_o \rangle$	whereas V_o and H_o represent B-operation names.
Body	$\langle K, I, U, O \rangle$	K is the predicate which constrains P, I is the B-invariant predicate, U is the B-substitution of initialisation, O maps $V_o \cup H_o$ to generalized substitutions: $O = \{o \mapsto \text{PRE } Q \text{ THEN } S \text{ END}\}$

Hidden variables and operations may occur as a result of assembly clauses USES and INCLUDES, essentially to ensure visibility rules, and to "mark" the shareable part of the component. Please note that since hidden variables may occur in the body, the invariant may reference them. Furthermore, operations may "call" other operations (visible or hidden). The consistency of a B-component is defined by the proof obligations of a B-machine, namely:

1- initialisation sets up the invariant $K \Rightarrow [U]I$
2- operations preserve the invariant $K \wedge I \wedge Q \Rightarrow [S]I$ for each $o \in V_o$
where $O(o) = \text{PRE } Q \text{ THEN } S \text{ END}$

Remind you that initialisation must set up all the variables. Note also that there is no proof obligation for hidden operations.

4.2 Abstract Machines and B-components

Let us examine a simple machine (without any relations with other machines), like the one of subsection 3.1. Such a syntactic module is mapped in a component. Component of M is constituted of the following elements (very obviously):

$$M = (P, \langle X, o\rangle, \langle \emptyset, \emptyset\rangle, \langle K, I, U, O\rangle)$$

Conversely, any valid B-component, where the set of hidden variables is empty, can be "decompiled" into a simple abstract machine (this can be a criterion to decide if an assembly of machines is complete or not): hidden operations are eliminated by replacing their name with the right part of their definition in the specification of visible operations.

4.3 Operations on B-components and their proof obligations

Throughout this section, we shall use the following notations, where B, B_1, B_2 denote bodies, C, C_1, C_2 denote valid components, and ΔC a tuple of component information:

B	$\langle K, I, U, O\rangle$
B_1	$\langle K_1, I_1, U_1, O_1\rangle$
B_2	$\langle K_2, I_2, U_2, O_2\rangle$
C	$(P, \langle V_x, V_o\rangle, \langle H_x, H_o\rangle, B)$
C_1	$(P_1, \langle V_{x_1}, V_{o_1}\rangle, \langle H_{x_1}, H_{o_1}\rangle, B_1)$
C_2	$(P_2, \langle V_{x_2}, V_{o_2}\rangle, \langle H_{x_2}, H_{o_2}\rangle, B_2)$
ΔC	$(P_2, \langle V_{x_2}, V_{o_2}\rangle, \langle H_{x_2}, H_{o_2}\rangle, B_2)$

$\otimes$: a new composition of substitutions. For the definition of union with implicit sharing of bodies, one needs to compose two operations with the same name. Let S_1 and S_2 be two substitutions, respectively modifying (without loss of generality) variables x, x_1 and x, x_2. $S_1 \otimes S_2$ must denote a substitution working on x, x_1, x_2, such that the effects on x_1 and x_2 are those of S_1 and S_2, just like a parallel composition of substitutions. Furthermore, its effect on the common variable x must be compatible with the effects on x of S_1 and S_2. Following the B-book, we define $S_1 \otimes S_2$ by its termination and before-after predicates:

$$trm(S_1 \otimes S_2) = trm(S_1) \wedge trm(S_2)$$
$$prd_{x,x_1,x_2}(S_1 \otimes S_2) = prd_{x,x_1}(S_1) \wedge prd_{x,x_2}(S_2)$$

So the normalized form of $S_1 \otimes S_2$ is:

$$S_1 \otimes S_2 = trm(S_1) \wedge trm(S_2) | @x', x'_1, x'_2.(prd_{x,x_1}(S_1) \wedge prd_{x,x_2}(S_2) \Longrightarrow x, x_1, x_2 := x', x'_1, x'_2)$$

Examples: 1- $(v :\in \{1,2,3\}) \otimes (v :\in \{2,3,4\}) \equiv (v :\in \{2,3\})$
2- $(v := 1) \otimes (v := 2) \equiv (v :\in \emptyset)$
($v :\in \emptyset$ is an unfeasible substitution)
3- $(\text{PRE } v < w \text{ THEN } v := v + 1 \text{ END}) \otimes (\text{PRE } v < w \text{ THEN } w := w - 1 \text{ END})$
$\equiv (\text{PRE } v < w \text{ THEN } v, w := v + 1, w - 1 \text{ END})$

When S_1 and S_2 work on distinct variables (it is the case for disjoint union), termination and before-after predicates become exactly those of parallel composition: $S_1 \otimes S_2 = S_1 || S_2$.

Lastly, an essential property of $\otimes$ is the following one. Let S_1 and S_2 be substitutions, P_1 and P_2 be predicates such that free variables of P_1 are set up by S_1, and those of P_2 by S_2 respectively. Then $[S_1]P_1 \wedge [S_2]P_2 \Rightarrow [S_1 \otimes S_2](P_1 \wedge P_2)$ holds.

Operations on bodies. Operations on bodies are defined as follows, where σ is a substitution and $O_1 \otimes O_2$ means the union of the two maps on operations, with merge (by $\otimes$) of operations with the same name:

Operation	Result
$\sigma(B)$	$\langle \sigma(K), \sigma(I), \sigma(U), \sigma(O) \rangle$
$B_1 \otimes B_2$	$\langle K_1 \wedge K_2, I_1 \wedge I_2, U_1 \otimes U_2, O_1 \otimes O_2 \rangle$
$B_1 \oplus B_2$	$\langle K_1 \wedge K_2, I_1 \wedge I_2, U_1 \|\| U_2, O_1 \cup O_2 \rangle$
$B_1 \rhd B_2$	$\langle K_1 \wedge K_2, I_1 \wedge I_2, U_1 ; U_2, O_1 \cup O_2 \rangle$

Notice that for enrichment, the initialisation U_2 and the operations O_2 may use and modify the variables $V_{x_1} \cup H_{x_1}$, so U_1 must be performed before U_2: the initialisation of the resulting body is the sequential composition of U_1 and U_2.

Operations on components. Operations on components are defined as follows (the last column indicates the presence of proof obligations):

Component	Signature	Body	P.O.
promote E_x **in** C	$(P, \langle V_x \cup E_x, V_o \rangle, \langle H_x - E_x, H_o \rangle)$	B	no
promote E_o **in** C	$(P, \langle V_x, V_o \cup E_o \rangle, \langle H_x, H_o - E_o \rangle)$	B	yes
hide E_x **in** C	$(P, \langle V_x - E_x, V_o \rangle, \langle H_x \cup E_x, H_o \rangle)$	B	no
hide E_o **in** C	$(P, \langle V_x, V_o - E_o \rangle, \langle H_x, H_o \cup E_o \rangle)$	B	no
instanciate C **by** σ	$(P - dom(\sigma), \langle V_x, V_o \rangle, \langle H_x, H_o \rangle)$	$\sigma(B)$	yes
rename C **by** σ	$(\sigma(P), \langle \sigma(V_x), \sigma(V_o) \rangle, \langle \sigma(H_x), \sigma(H_o) \rangle)$	$\sigma(B)$	no
$C_1 \otimes C_2$	$(P_1 \cup P_2, \langle V_{x_1} \cup V_{x_2}, V_{o_1} \cup V_{o_2} \rangle, \langle H_{x_1} \cup H_{x_2}, H_{o_1} \cup H_{o_2} \rangle)$	$B_1 \otimes B_2$	yes
$C_1 \oplus C_2$	$(P_1 \cup P_2, \langle V_{x_1} \cup V_{x_2}, V_{o_1} \cup V_{o_2} \rangle, \langle H_{x_1} \cup H_{x_2}, H_{o_1} \cup H_{o_2} \rangle)$	$B_1 \oplus B_2$	no
enrich C_1 **by** ΔC	$(P_1 \cup P_2, \langle V_{x_1} \cup V_{x_2}, V_{o_1} \cup V_{o_2} \rangle, \langle H_{x_1} \cup H_{x_2}, H_{o_1} \cup H_{o_2} \rangle)$	$B_1 \rhd B_2$	yes

Proof obligation of: promote E_o **in** C. Promoving an operation requires that it preserves the invariant:

$$K \wedge I \wedge Q \Rightarrow [S]I \text{ for each } o \in E_o \text{ where } O(o) = \text{PRE } Q \text{ THEN } S \text{ END}$$

Proof obligation of: instanciate C **by** σ. The instanciation of parameters must satisfy the constraints:

$$\sigma(K)$$

Proof obligations of: $C_1 \otimes C_2$.

1- $K_1 \wedge K_2 \wedge I_1 \wedge I_2 \wedge Q \Rightarrow [S]I_2$ for each $o \in V_{o_1} - V_{o_2}$
2- $K_1 \wedge K_2 \wedge I_1 \wedge I_2 \wedge Q \Rightarrow [S]I_1$ for each $o \in V_{o_2} - V_{o_1}$
3- $K_1 \wedge K_2 \wedge I_1 \wedge I_2 \wedge Q \Rightarrow [S](I_1 \wedge I_2)$ for each $o \in V_{o_1} \cap V_{o_2}$
where $O(o) = \text{PRE } Q \text{ THEN } S \text{ END}$

There is no proof obligation that initialisation sets up the invariant, because $K_1 \Rightarrow [U_1]I_1$, $K_2 \Rightarrow [U_2]I_2$ (initialisations come from consistent components), and all free variables of I_1 are set up by U_1 and those of I_2 by U_2, therefore $K_1 \wedge K_2 \Rightarrow [U_1 \otimes U_2](I_1 \wedge I_2)$ holds, thanks to the main property of $\otimes$.

The first and second proof obligations take into account that independent operations are defined in consistent components, so they preserve their invariants.

The third proof obligation is mandatory. For instance, let I_1 and I_2 be $v \leq w$, $O_1(o) =$ PRE $v < w$ THEN $v := v + 1$ END, $O_2(o) =$ PRE $v < w$ THEN $w := w - 1$ END. Each operation preserves its invariant, but not their merge.

Proof obligations of enrich C_1 **by** ΔC.

1- $K_1 \wedge K_2 \Rightarrow [U_1; U_2](I_1 \wedge I_2)$
2- $K_1 \wedge K_2 \wedge\ I_1 \wedge I_2\ \wedge\ Q \Rightarrow [S]I_2$ for each $o \in V_{o_1}$
3- $K_1 \wedge K_2 \wedge\ I_1 \wedge I_2\ \wedge\ Q \Rightarrow [S](I_1 \wedge I_2)$ for each $o \in V_{o_2}$
where $O(o) =$ PRE Q THEN S END

5 B Assembly Clauses Revisited

In this section, the effect of clauses USES and INCLUDES as component operations is described as a composition of primitive operations. So a consistent B-component is obtained for each possible construction. However, by this way, some restrictions are introduced and some valid constructions in the B method are no more representable in this formalism.

5.1 Clause USES seen as a component operation.

Consider the following USES construction:

```
MACHINE
    M(P)
USES
    N1, ... Nn
CONSTRAINTS
    K
VARIABLES
    X
INVARIANT
    I
INITIALISATION
    U
OPERATIONS
    O
END
```

Let $C_i = (P_i, \langle V_{xi}, V_{o_i}\rangle, \langle H_{x_i}, H_{o_i}\rangle, \langle K_i, I_i, U_i, O_i\rangle)$ be the components associated with machines N_i, with $P \cap \bigcup_{1 \leq i \leq n} P_i = \emptyset$. If it is not the case P can be renamed. The construction of the component associated with the machine M can be built by the following sequence of basic component operations:

1. **hide** each visible operation of component C_i. By this way, operations of the used machines cannot be referenced in the local enrichment of the using machine.
2. **build the union** of the used components. In this step, union is not disjoint because used components can already share some variables. The resulting component is now parametrized by all parameters of C_i.
3. **enrich** the obtained component with the local declarations of the using machine.

4. **hide** the visible variables of the used components. This step describes the non-transitivity of the USES clause.

   Thus we obtain:

(1) for each C_i do $C_{i,1} =$	**hide** V_{o_i} **in** C_i		no proof obligation
(2)	$A =$	$C_{1,1} \otimes \ldots \otimes C_{n,1}$	
(3)	$B =$	**enrich** A **by** $(P, \langle X, dom(O)\rangle, \langle\emptyset,\emptyset\rangle, \langle K, I, U, O\rangle)$	
(4)	$C =$	**hide** $\bigcup_{1\leq i\leq n} V_{x_i}$ **in** B	no proof obligation

In step 2 no proof obligation is needed because no operation is visible. In step 3 proof obligations are limited to show that all operations defined in O preserve I, because they do not modify shared variables and thus $\bigwedge_{1\leq i\leq n} I_i$ is preserved.

The proof obligations resulting from this component elaboration are different from the method B ones: B-component consistency imposes that all operations in $dom(O)$ verify the invariants $\bigwedge_{1\leq i\leq n} I_i$ *and* also I. For instance in machine Ma_3 of example 2, a new proof obligation is added: op_3 must preserve the formula $z = x + y$.

The resulting proof obligations, imposed by this construction, are comparable with those of an INCLUDES construction in which included machines are not instantiated and no operation is promoted. Moreover the parameters of the using machine are extended by the parameters of the used machines. In this sense we bring the gap between the clauses USES and INCLUDES.

The modification proposed here is not insignificant: it does not only consist in proving some formulae sooner, but it constrains to prove them independently of the final instantiations of the used machine. Thus some valid constructions in the B method are, here, rejected. But from a methodological point of view, such a restriction seems a good one: why would we locally define an operation which is only correct for some instantiations ? Why could not we delay the definition of this operation in the context of its complete validation ?

5.2 INCLUDES primitive and component operations.

Consider the following INCLUDES construction:

```
MACHINE
    M(P)
INCLUDES
    N_1(E_1), ... N_n(E_n)
PROMOTES
    o          /* o ⊆ ⋃_{1≤i≤n} O_i */
CONSTRAINTS
    K
VARIABLES
    X
INVARIANT
    I
INITIALISATION
    U
OPERATIONS
    O
END
```

Let $C_i = (P_i, \langle V_{x_i}, V_{o_i}\rangle, \langle H_{x_i}, H_{o_i}\rangle, \langle K_i, I_i, U_i, O_i\rangle)$ be the components associated with machines N_i, and $P \cup \bigcap_{1\leq i\leq n} P_i = \emptyset$. In the language B only two cases of construction are valid:

– (1) each C_i are simple components, i.e components without hidden variables.

– (2) the construction is a closure, i.e. all hidden variables are visible in another component:

$$\bigcup_{1\leq i\leq n} H_{x_i} \subseteq \bigcup_{1\leq j\leq n} V_{x_j}$$

The first case is a subcase of (2). The resulting component is obtained in the following way:

1. **parametrize** all components C_i by P.
2. **instantiate** each obtained component by σ_i, which corresponds to the instantiation of parameters of N_i by E_i. We suppose that the sets $dom(\sigma_i)$ are disjoint, (if it is not the case we may apply a renaming substitution).
3. **instantiate** each resulting component by σ'_i with :
   – $\sigma = \bigcup_{1\leq i\leq n} \sigma_i$,
   – $\sigma'_i = \sigma \cap (id(P_i) - dom(\sigma_i))$.

   Because some parameters of C_i come from used components, σ'_i is necessary to obtain the right instantiation.
4. **hide** operations of included components which are not promoted in the current construction.
5. **promote** the hidden variables.
6. **build the union** of instantiated components. By this operation shared variables are merged.
7. **enrich** the obtained component with the local declarations of the including machine.

This construction is described by the following components elaboration:

(1) for each C_i do $C_{i,1} =$ **enrich** C_i **by** $(P, \langle\emptyset,\emptyset\rangle, \langle\emptyset,\emptyset\rangle, \langle K, true, skip, \emptyset\rangle)$
(2) for each $C_{i,1}$ do $C_{i,2} =$ **instantiate** $C_{i,1}$ **by** σ_i
(3) for each $C_{i,2}$ do $C_{i,3} =$ **instantiate** $C_{i,2}$ **by** σ'_i
(4) for each $C_{i,3}$ do $C_{i,4} =$ **hide** $V_{o_i} - o$ **in** $C_{i,3}$
(5) for each $C_{i,4}$ do $C_{i,5} =$ **promote** $\bigcup_{1\leq i\leq n} H_{x_i}$ **in** $C_{i,4}$
(6) $B = C_{1,5} \otimes \ldots \otimes C_{n,5}$
(7) $C =$ **enrich** B **by** $(\emptyset, \langle X, dom(O)\rangle, \langle\emptyset,\emptyset\rangle, \langle K, I, U, O'\rangle)$
where O' stands for O with calls of operations replaced by their definition.

In step 2, proofs obligations correspond to those of the instantiation operations. Let us consider the case of a simple INCLUDES construction:

– step 6: the shared part is restricted to the parameter P. Because variables are disjoint, so proof obligations are not needed.
– step 7: if we consider the syntactic restrictions of the B method, proof obligations are restricted to show that operations in $dom(O) \cup o$ preserve I. By hypothesis, o preserve $\bigwedge_{1\leq i\leq n} I_i$, and operations in $dom(O)$ also preserve this invariant, because they modify variables of the included machines in a controlled way (syntactic restrictions).

In this case proof obligations are the same as in the B method. Now, in the case of a closure construction, proof obligations are:

– step 6: because variables of the same name are shared, all operations which modify some of these variables must be proved again. These operations can be defined only in the shared machines in which these variables are introduced. Thus only operations coming from used machines must be proved again.
– step 7: no simplification arises. All promoted and locally defined operations must verify the invariants I and $\bigwedge_{1\leq i\leq n} I_i$.

In this case we obtain fewer proof obligations than in the B method. Some proofs which are delayed in the B method are now proved when the USES construction takes place.

6 Conclusion

In conclusion, we want to discuss the lessons learnt in using component primitives to define assembly clauses of the B language. A major point of the work is that we have tried to perform altogether incremental machine construction and proofs of consistency of the elaborated machines. By this way, each intermediary step is a reusable component and a further usage of a component does not depend on the subcomponents construction. When considering the B assembly clauses, we founded that they are not "orthogonal" with this respect. Actually, there are three situations in an assembly of machines. They are:

– a machine INCLUDES independent machines (independent INCLUDES).
– a machine INCLUDES a closure of shared machines (sharing INCLUDES).
– a machine USES another machine (using).

These three cases can be compared following three aspects. The first one is the ability to build the text of a new simple machine "equivalent" to the assembly of machines (regularity); the second one is the fact that the proofs done at the level of a machine ensure the soundness of this machine in any valid context (integrity); and the last one concerns the fact that auxiliary proofs are not needed to validate the accesses to some submachines (locality). For instance, in the INCLUDES of independent machines, locality is ensured by syntactic restrictions (modification of submachine variables by operation calls only and no call of operations of the same machine in both parts of a parallel substitution). The regularity aspect is satisfied only in the case of the simple independent INCLUDES clause, or when including a closure of shared machines. We have seen that a machine containing a USES clause cannot be equivalent to a simple machine. With respect to integrity, a using machine can be proved as preserving its own invariant in some closure and not preserving it in another one, depending on the operations which are promoted in the final step. So, a using machine does not satisfy the integrity criterion. One can notice that if the meaning of an assembly of machines is defined by a flat (simple) machine and if the consistency conditions of the assembly clauses imply the consistency of the resulting machine, then integrity of machine assemblies is equivalent to integrity of simple machines. Moreover, integrity and locality are related, because if all the constructions satisfy the integrity criterion, then it is not needed to do non local proofs somewhere to ensure consistency of the whole. For a specification language, the aim is that any complex construction of modules is sound. This aim can be achieved by syntactic restrictions which preserve integrity of subparts (like it is done for an independent INCLUDES). But if a more flexible strategy and a better expressiveness is prefered, then integrity can be not preserved and soundness is only guaranteed by some non local proofs. In summary, the assembly constructions of B satisfy:

	regularity	integrity	locality
independent INCLUDES	yes	yes	yes
sharing INCLUDES	yes	yes	no
using	no	no	yes

Our revisited INCLUDES and USES primitives tend to satisfy regularity and integrity conditions. The price to be paid is that some constructs which are valid in the B method will be rejected in our approach because consistency conditions of a using machine are stronger in this paper than in the B language.

At last, we hope that the study initiated in this paper can help to understand some subtle aspects of the assembly clauses of the B language.

References

[Abr96] J. R. Abrial. *B Book.* Cambridge University Press, 1996.

[BE95] D. Bert and R. Echahed. Multiparadigm logic programming : the case of the language LPG. Technical report, IMAG-LGI, 1995.

[BG77] R. M. Burstall and J. A. Goguen. Putting theories together to make specifications. In *Proc. of 5th Int. Joint Conference on Artificial Intelligence, Cambridge, Mass.*, pages 1045–1052, 1977.

[BHK90] J. A. Bergstra, J. Heering, and P. Klint. Module algebra. *J. ACM*, 37(2):335–372, 1990.

[BL91] D. Bert and Ch. Lafontaine. Integration of semantical verification conditions in a specification langage definition. In *Proc. of the 2nd Conf. on Algebraic Methodology and Software Technology (AMAST-91)*, pages 467–477. Springer-Verlag, 1991.

[EFH83] H. Ehrig, W. Fey, and H. Hansen. ACT ONE: an algebraic specification language with two levels of semantics. Technical report 83-03, TU Berlin, 1983.

[Fey88] W. Fey. Pragmatics, Concepts, Syntax, Semantics and Correctness Notions of ACT TWO: an algebraic module specification and interconnection language. Technical report 88-26, TU Berlin, 1988.

[GB90] J. A. Goguen and R. M. Burstall. Institutions: Abstract Model Theory for Specification and Programming. Research Report ECS-LFCS-90-106, Lab. for Foundations of Computer Science, Univ. of Edinburgh, Jan. 1990.

[Ori96] C. Oriat. Etude des spécifications modulaires : constructions de colimites finies, diagrammes, isomorphismes. Doctorat d'Université, INPG, Grenoble, Janvier 1996.

[SJ95] Y. V. Srinivas and R. Jüllig. SPECWARE: Formal support for composing software. In *Proc. of the Conf. on Mathematics of Program Construction*, 1995.

[Spi88] M. Spivey. *Understanding Z : a specification language and its formal semantics.* Cambridge University Press, 1988.

[ST88] D. Sannella and A. Tarlecki. Towards formal development of programs from algebraic specifications : Implementations revisited. *Acta Informatica*, 25:233–281, 1988.

[Wir86] M. Wirsing. Structured Algebraic Specifications: A Kernel Language. *Theoretical Computer Science*, 42:123–249, 1986.