A Study on Components and Assembly Primitives in B

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

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 or 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 \( \text{vis}(C) \). Visible identifiers are divided into parameters: \( \text{par}(C) \) and declared identifiers: \( \text{dec}(C) \). In a component, a part of the information can be hidden, so it is sometimes needed to deal with a hidden signature, called \( \text{hid}(C) \).

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 explicitly 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 “\( \Sigma \)-body \( \beta \)” to speak of an environment \( \beta \) on the identifiers \( \Sigma \). Conditions of well-definedness of a component can be formalized in this notation. They are validity or consistency conditions depending on the signature \( \Sigma \).

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

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

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

\[
C = (P, V, H, B)
\]

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

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 \( \Sigma_1 \) be \((P_1, V_1, H_1)\) and \( \Sigma_2 \) 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 \( \Sigma_1, \Sigma_2 \), a signature mapping \( \sigma \in \Sigma_1 \rightarrow \Sigma_2 \) also denoted \( \Sigma_1 \xrightarrow{\sigma} \Sigma_2 \) is a correspondence between the identifiers of \( \Sigma_1 \) and those of \( \Sigma_2 \) 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 \( \Sigma \) 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 \( \Sigma_1 \)-body and \( B_2 \) be a \( \Sigma_2 \)-body, then we denote by \( B_1 \oplus B_2 \) 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 \( \Sigma_1 \) and \( \Sigma_2 \) are disjoint, then the operation merge is equivalent to the union of the two bodies, denoted \( B_1 \cup B_2 \). 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 \( \Sigma_1 \)-body and \( \Sigma_1 \subseteq \Sigma_2 \). Assume that \( B_2 \) is a set of body information on \( \Sigma_2 \), then \( B_1 \triangleright B_2 \) is a \( \Sigma_2 \)-body where \( B_1 \) is enriched by \( B_2 \). The operation \( \triangleright \) 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 \( B \)-body.

At last, let \( \sigma \) be a signature mapping \( \Sigma_1 \rightarrow \Sigma_2 \) and \( B \) be a \( \Sigma_1 \)-body, then \( \sigma(B) \) is a \( \Sigma_2 \)-body obtained by the application of the mapping \( \sigma \) to the body \( B \). The result of the application is that each occurrence of a name \( n \) in \( B \) such that \( n \in \text{dom}(\sigma) \) 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 explicitly 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:

\[
\text{promote } E \text{ in } C = (P, V \cup E, H - E, B)
\]

**Hiding.** Hiding is the operation symmetric 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:

\[
\text{hide } E \text{ in } C = (P, V - E, H \cup E, B)
\]

**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 \( \sigma \) be a substitution of the form \( \{x_i \mapsto v_i\} \) for \( i \in [1, k] \), where \( \text{dom}(\sigma) \subseteq P \), 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:

\[
\text{instantiate } C \text{ by } \sigma = (P - \text{dom}(\sigma), V, H, \sigma(B))
\]

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 \( \Sigma_1 \rightarrow \Sigma_2 \), 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 \( P_1 \rightarrow P_2 \), \( V_1 \rightarrow V_2 \), \( H_1 \rightarrow H_2 \). 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 \( \Sigma_1 \rightarrow \Sigma_2 \), then:

\[
\text{rename } C \text{ by } \sigma = (\Sigma_2, \sigma(B))
\]

**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 $C_1 = (\Sigma_1, B_1)$ and let $C_2 = (\Sigma_2, B_2)$, 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 symmetrically for the identifiers of $C_2$ then:

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

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:

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

**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 $C = (\Sigma, B_1)$ 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:

$$\text{enrich } C \text{ by } \Delta C = (\Sigma_1 \cup \Sigma_2, B_1 \oplus B_2)$$

### 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:

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

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 find 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 \textsc{sets}, \textsc{constants}, \textsc{properties}, and the conditions associated to the parameters (non emptiness and finiteness of the sets, etc).
An abstract machine is consistent if, and only if the following conditions (called \textit{proof obligations}) hold:

1. initialisation sets up the invariant: $K \implies [U]I$
2. operations preserve the invariant: $K \land I \land Q \implies [S]I$ for each operation $o \in \text{dom}(O)$

(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 \textsc{uses} and \textsc{includes} allow us to elaborate union of abstract machines, combined with enrichments: the clause \textsc{includes} creates local copies of instantiated abstract machines, whereas the clause \textsc{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:

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 \textit{closure of a \textsc{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 \textsc{includes} when no clause \textsc{uses} appears in the included machines, then the effect of the clause \textsc{uses} and the treatment of the closure.

\subsection*{3.2 Description of the clause \textsc{includes} effect}

A machine with a clause \textsc{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.
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 $\text{dom}(O_2)$ preserve the invariant $I_1$, and $a \text{ fortiori} P_1$. Thus the local proofs are restricted to show that operations offered by the machine $M_2$, i.e. $\text{dom}(O_2) \cup o_1$, preserve $I_2$.

**Example 1.**

\[\begin{array}{l}
\text{MACHINE } M_1 \{ P_1 \} \\
\text{CONSTRAINTS } K_1 \\
\text{VARIABLES } X_1 \\
\text{INVARIANT } I_1 \\
\text{INITIALISATION } U_1 \\
\text{OPERATIONS } O_1 \\
\text{END}
\end{array}\]

$M_1$ is included in $M_2$:

\[\begin{array}{l}
\text{MACHINE } M_2 \{ P_2 \} \\
\text{INCLUDES } M_1 \{ E_1 \} \\
\text{PROMOTES } o_1 \\
\text{/* operations of } M_1 \text{ which become operations of } M_2 */ \\
\text{CONSTRAINTS } K_2 \\
\text{VARIABLES } X_2 \\
\text{INVARIANT } I_2 \\
\text{INITIALISATION } U_2 \\
\text{OPERATIONS } O_2 \\
\text{END}
\end{array}\]

The resulting abstract machine equivalent to $M_2$ is:

\[\begin{array}{l}
\text{MACHINE } M_2 \{ P_2 \} \\
\text{CONSTRAINTS } K_2 \\
\text{VARIABLES } X_2, X_1 \\
\text{INVARIANT } I_2 \wedge I_1 \\
\text{INITIALISATION } U_1 \cup U_2 \\
\text{OPERATIONS } O_2, o_1 \\
\text{END}
\end{array}\]

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:

\[\begin{array}{l}
\text{MACHINE } M_{a_1} \{ \} \\
\text{VARIABLES } x, y \\
\text{INVARIANT } x \in \text{NAT} \wedge y \in \text{NAT} \wedge x < y \\
\text{INITIALISATION } x, y := 0, 1 \\
\text{OPERATIONS } op_1 = S_1; op_2 = S_2 \\
\text{END}
\end{array}\]

$M_{a_1}$ is included in $M_{a_2}$:

\[\begin{array}{l}
\text{MACHINE } M_{a_2} \{ \} \\
\text{INCLUDES } M_{a_1} \\
\text{PROMOTES } op_1 \\
\text{VARIABLES } z \\
\text{INVARIANT } z \in \text{NAT} \wedge z = x + y \\
\text{INITIALISATION } z := x + y \\
\text{OPERATIONS } op_3 = S_3 \\
\text{END}
\end{array}\]

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

\(^2\) $S_1||S_2$ is well-defined if, and only if, the sets of variables modified by $S_1$ and $S_2$ are disjoint.
- \( op_1 \) and \( op_3 \) preserve the formula \( x \in \text{NAT} \land y \in \text{NAT} \land x < y \land z \in \text{NAT} \land z = x + y \);
- \( op_2 \) preserves the formula \( x \in \text{NAT} \land y \in \text{NAT} \land x < y \).

### 3.3 Description of the clause USES effect

The construction \textsc{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:

<table>
<thead>
<tr>
<th>MACHINE</th>
<th>( M_1(P_1) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONSTRAINTS</td>
<td>( K_1 )</td>
</tr>
<tr>
<td>VARIABLES</td>
<td>( X_1 )</td>
</tr>
<tr>
<td>INVARIANT</td>
<td>( I_1 )</td>
</tr>
<tr>
<td>INITIATION</td>
<td>( U_1 )</td>
</tr>
<tr>
<td>OPERATIONS</td>
<td>( O_1 )</td>
</tr>
<tr>
<td>END</td>
<td></td>
</tr>
</tbody>
</table>

\( M_1 \) is used by \( M_2 \):

<table>
<thead>
<tr>
<th>MACHINE</th>
<th>( M_2(P_2) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>USES</td>
<td>( M_1 )</td>
</tr>
<tr>
<td>CONSTRAINTS</td>
<td>( K_2 )</td>
</tr>
<tr>
<td>VARIABLES</td>
<td>( X_2 )</td>
</tr>
<tr>
<td>INVARIANT</td>
<td>( I_2 )</td>
</tr>
<tr>
<td>INITIATION</td>
<td>( U_2 )</td>
</tr>
<tr>
<td>OPERATIONS</td>
<td>( O_2 )</td>
</tr>
<tr>
<td>END</td>
<td></td>
</tr>
</tbody>
</table>

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 \textsc{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 \textsc{includes} for different machines.

**Example 2.**

<table>
<thead>
<tr>
<th>MACHINE</th>
<th>( Ma_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARIABLES</td>
<td>( z )</td>
</tr>
<tr>
<td>USES</td>
<td>( Ma_1 )</td>
</tr>
<tr>
<td>INVARIANT</td>
<td>( z \in \text{NAT} \land z = x + y )</td>
</tr>
<tr>
<td>INITIATION</td>
<td>( z := x + y )</td>
</tr>
<tr>
<td>OPERATIONS</td>
<td>( op_3 = S_3 )</td>
</tr>
<tr>
<td>END</td>
<td></td>
</tr>
</tbody>
</table>

\( Ma_1 \) and \( Ma_3 \) are included into \( Ma_4 \):

<table>
<thead>
<tr>
<th>MACHINE</th>
<th>( Ma_4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>INCLUDES</td>
<td>( Ma_1, Ma_3 )</td>
</tr>
<tr>
<td>PROMOTES</td>
<td>( op_1, op_3 )</td>
</tr>
<tr>
<td>END</td>
<td></td>
</tr>
</tbody>
</table>
In terms of consistency we have:

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

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 revisit the clauses uses and includes as component operations. A comparison, 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.

<table>
<thead>
<tr>
<th>Name of the field</th>
<th>Content</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>$P$</td>
<td>List of the parameters names.</td>
</tr>
<tr>
<td>Visible names</td>
<td>$(V_o, V_x)$</td>
<td>The meta-variables $V_o$ and $H_x$ mean B-variable names</td>
</tr>
<tr>
<td>Hidden names</td>
<td>$(H_o, H_x)$</td>
<td>whereas $V_o$ and $H_x$ represent B-operation names</td>
</tr>
<tr>
<td>Body</td>
<td>$(K, I, U, O)$</td>
<td>$K$ 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; $O = { s \mapsto \text{pre Q then S end} }$</td>
</tr>
</tbody>
</table>
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 $K \Rightarrow \exists U \forall I$
2. operations preserve the invariant $K \land I \land Q \Rightarrow [S]I$ for each $o \in V_o$

where $O(o) = \text{PRE} Q \Rightarrow \text{THEN} S \text{ END}$

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):

$$M = (P, \{X, o\}, \{\emptyset, \emptyset\}, \{K, I, U, O\})$$

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 $\Delta C$ a tuple of component information:

<table>
<thead>
<tr>
<th>$B$</th>
<th>$(K, I, U, O)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B_1$</td>
<td>$(K_1, I_1, U_1, O_1)$</td>
</tr>
<tr>
<td>$B_2$</td>
<td>$(K_2, I_2, U_2, O_2)$</td>
</tr>
<tr>
<td>$C$</td>
<td>$(P, {V_o, V_o}, {H_o, H_o}, B)$</td>
</tr>
<tr>
<td>$C_1$</td>
<td>$(P_1, {V_o, V_o}, {H_o, H_o}, B_1)$</td>
</tr>
<tr>
<td>$C_2$</td>
<td>$(P_2, {V_o, V_o}, {H_o, H_o}, B_2)$</td>
</tr>
<tr>
<td>$\Delta C$</td>
<td>$(P_2, {V_o, V_o}, {H_o, H_o}, B_2)$</td>
</tr>
</tbody>
</table>

⊗: 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 $x, x_2$. $S_1 \circ S_2$ 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 $S_1 \circ S_2$ by its termination and before-after predicates:

$$\text{trm}(S_1 \circ S_2) = \text{trm}(S_1) \land \text{trm}(S_2)$$

$$\text{prd}_{x,x_1}(S_1 \circ S_2) = \text{prd}_{x,x_1}(S_1) \land \text{prd}_{x,x_2}(S_2)$$

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

$$S_1 \circ S_2 = \text{trm}(S_1) \land \text{trm}(S_2)[\forall x', x_1, x_2, (\text{prd}_{x,x_1}(S_1) \land \text{prd}_{x,x_2}(S_2) \Rightarrow x, x_1, x_2 := x', x_1', x_2')$$
Examples: 1. \( (v \in \{1, 2, 3\} \cap (v \in \{2, 3, 4\}) \equiv (v \in \{2, 3\}) \)
2. \((v \equiv 1) \cap (v \equiv 2) \equiv (v \equiv \emptyset)\)
   \((v \equiv \emptyset)\) is an unfeasible substitution
3. \((\text{pre } v < w \text{ then } v := v + 1 \text{ end}) \cap (\text{pre } v < w \text{ then } w := w - 1 \text{ end}) \equiv (\text{pre } v < w \text{ then } v, w := v + w - 1 \text{ end})\)

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: \( S_1 \circ S_2 = S_1 \parallel S_2 \).

Lastly, an essential property of \( \circ \) 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 \([S_1]P_1 \land [S_2]P_2 \Rightarrow [S_1 \circ S_2](P_1 \land P_2)\) holds.

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \sigma(B) )</td>
<td>( \langle \sigma(K), \sigma(I), \sigma(U), \sigma(O) \rangle )</td>
</tr>
<tr>
<td>( B_1 \circ B_2 )</td>
<td>( (K_1 \wedge K_2, I_1 \wedge I_2, U_1 \circ U_2, O_1 \circ O_2) )</td>
</tr>
<tr>
<td>( B_1 \triangleright B_2 )</td>
<td>( (K_1 \wedge K_2, I_1 \wedge I_2, U_1; U_2, O_1 \circ O_2) )</td>
</tr>
</tbody>
</table>

Notice that for enrichment, the initialisation \( U_2 \) and the operations \( O_2 \) may use and modify the variables \( V_{x_1} \cup H_{x_2} \), 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):

<table>
<thead>
<tr>
<th>Component</th>
<th>Signature</th>
<th>Body</th>
<th>P.O.</th>
</tr>
</thead>
<tbody>
<tr>
<td>promote ( E_a ) in ( C )</td>
<td>( (P, \langle V_x \cup E_a, V_o \rangle, \langle H_x - E_x, H_o \rangle) )</td>
<td>( B )</td>
<td>no</td>
</tr>
<tr>
<td>promote ( E_a ) in ( C )</td>
<td>( (P, \langle V_x, V_o \cup E_a \rangle, \langle H_x, H_o - E_x \rangle) )</td>
<td>( B )</td>
<td>yes</td>
</tr>
<tr>
<td>hide ( E_a ) in ( C )</td>
<td>( (P, \langle V_x - E_x, V_o \rangle, \langle H_x \cup E_x, H_o \rangle) )</td>
<td>( B )</td>
<td>no</td>
</tr>
<tr>
<td>hide ( E_a ) in ( C )</td>
<td>( (P, \langle V_x, V_o - E_a \rangle, \langle H_x \cup H_o \cup E_x \rangle) )</td>
<td>( B )</td>
<td>no</td>
</tr>
<tr>
<td>instance ( C ) by ( \sigma )</td>
<td>( (P - \text{dom}(\sigma), \langle V_x, V_o \rangle, \langle H_x \rangle) )</td>
<td>( \sigma(B) )</td>
<td>yes</td>
</tr>
<tr>
<td>rename ( C ) by ( \sigma )</td>
<td>( (\sigma(P), \sigma(V_x), \sigma(V_o)), \sigma(H_x), \sigma(H_o)) )</td>
<td>( \sigma(B) )</td>
<td>no</td>
</tr>
<tr>
<td>( C_1 \circ C_2 )</td>
<td>( (P_1 \cup P_2, \langle V_x, \cup V_{x_2}, V_o \cup V_{o_2}, \langle H_x, \cup H_{x_2}, H_o, \cup H_{o_2} \rangle \rangle, \langle B_1 \circ B_2 \rangle) )</td>
<td>( B_1 \circ B_2 )</td>
<td>yes</td>
</tr>
<tr>
<td>( C_1 \circ C_2 )</td>
<td>( (P_1 \cup P_2, \langle V_x, \cup V_{x_2}, V_o \cup V_{o_2}, \langle H_x, \cup H_{x_2}, H_o, \cup H_{o_2} \rangle \rangle, \langle B_1 \circ B_2 \rangle) )</td>
<td>( B_1 \circ B_2 )</td>
<td>no</td>
</tr>
<tr>
<td>enrich ( C_1 ) by ( \Delta C )</td>
<td>( (P_1 \cup P_2, \langle V_x, \cup V_{x_2}, V_o, \cup V_{o_2}, \langle H_x, \cup H_{x_2}, H_o, \cup H_{o_2} \rangle \rangle, \langle B_1 \circ B_2 \rangle) )</td>
<td>( B_1 \circ B_2 )</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Proof obligation of:** promote \( E_a \) in \( C \). Promoting an operation requires that it preserves the invariant:

\( K \wedge I \wedge Q \Rightarrow [S]I \) for each \( o \in E_a \) where \( O(o) = \text{pre } Q \text{ then } S \text{ end} \)

**Proof obligation of:** instance \( C \) by \( \sigma \). The instanciation of parameters must satisfy the constraints:

\( \sigma(K) \)

**Proof obligations of:** \( C_1 \circ C_2 \).

1. \( K_1 \wedge K_2 \wedge I_1 \wedge I_2 \wedge Q \Rightarrow [S]I_2 \) for each \( o \in V_{x_1} - V_{x_2} \)
2. \( K_1 \wedge K_2 \wedge I_1 \wedge I_2 \wedge Q \Rightarrow [S]I_1 \) for each \( o \in V_{x_2} - V_{x_1} \)
3. \( K_1 \wedge K_2 \wedge I_1 \wedge I_2 \wedge Q \Rightarrow [S](I_1 \wedge I_2) \) for each \( o \in V_{x_1} \cap V_{x_2} \)

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 $K_1 \land K_2 \Rightarrow [U_1 \otimes U_2](I_1 \land I_2)$ 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 $v \leq w, O_1(o) = \text{pre } v < w$ then $v := v + 1$ end, $O_2(o) = \text{pre } v < w$ then $w := w - 1$ end. Each operation preserves its invariant, but not their merge.

**Proof obligations of enrich $C_1$ by $\Delta C$.**

1. $K_1 \land K_2 \Rightarrow [U_1; U_2](I_1 \land I_2)$
2. $K_1 \land K_2 \land I_1 \land I_2 \land Q \Rightarrow [S]I_2$ for each $o \in V_o$
3. $K_1 \land K_2 \land I_1 \land I_2 \land Q \Rightarrow [S](I_1 \land I_2)$ for each $o \in V_{o2}$
   where $O(o) = \text{pre } Q$ then $S$ end

**5 B Assembly Clauses Revisited**

In this section, the effect of clauses $\text{USES}$ and $\text{INCLUDES}$ 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 $\text{USES}$ seen as a component operation.**

Consider the following uses construction:

```
MACHINE M(P)
USES N_1, ..., N_n
CONSTRAINTS K
VARIABLES X
INVARIANT I
INITIALISATION U
OPERATIONS O
END
```

Let $C_i = (P_i, \langle V_{pi}, V_{si} \rangle, \langle H_{pi}, H_{si} \rangle, \langle K_i, I_i, U_i, O_i \rangle)$ 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 \( C_{i,1} = \text{hide } V_{o_i} \text{ in } C_i \) no proof obligation

(2) \( A = C_{1,1} \otimes \ldots \otimes C_{n,1} \)

(3) \( B = \text{enrich } A \text{ by } \{ P, \{ X, \text{dom}(O) \}, \{ \emptyset, \emptyset \}, \{ K, I, U, O \} \} \)

(4) \( C = \text{hide } \bigcup_{1 \leq i \leq n} V_{o_i} \text{ in } B \) 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 \( \text{dom}(O) \) verify the invariants \( \bigwedge_{1 \leq i \leq n} K_i \) 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:

```plaintext
MACHINE M(P)
INCLUDES N_1(E_1), \ldots, N_n(E_n)
PROMOTES o /* o \subseteq \bigcup_{1 \leq i \leq n} O_i */
CONSTRAINTS K
VARIABLES X
INVARIANT I
INITIALISATION U
OPERATIONS O
END
```

Let \( C_i = (P_i, \{ V_{x_i}, V_{o_i} \}, \{ H_{x_i}, H_{o_i} \}, \{ K_i, I_i, U_i, O_i \}) \) 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. **instatiate** each obtained component by \( \sigma_i \), which corresponds to the instantiation of parameters of \( N_i \) by \( E_i \). We suppose that the sets \( \text{dom}(\sigma_i) \) are disjoint, (if it is not the case we may apply a renaming substitution).
3. **instatiate** each resulting component by \( \sigma'_i \) with:
   - \( \sigma = \bigcup_{1 \leq i \leq n} \sigma_i \),
   - \( \sigma'_i = \sigma \cap (\text{id}(P_i) - \text{dom}(\sigma_i)) \).

   Because some parameters of \( C_i \) come from used components, \( \sigma'_i \) 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 \( C_{i,1} = \text{enrich } C_i \text{ by } (P, (\emptyset, \emptyset), (\emptyset, \emptyset), (K, \text{true}, \text{skip}, \emptyset)) \)
2. for each \( C_{i,1} \) do \( C_{i,2} = \text{instatiate } C_{i,1} \text{ by } \sigma_i \)
3. for each \( C_{i,2} \) do \( C_{i,3} = \text{instatiate } C_{i,2} \text{ by } \sigma'_i \)
4. for each \( C_{i,3} \) do \( C_{i,4} = \text{hide } V_{x_i} - o \text{ in } C_{i,3} \)
5. for each \( C_{i,4} \) do \( C_{i,5} = \text{promote } \bigcup_{1 \leq i \leq n} H_{x_i} \text{ in } C_{i,4} \)
6. \( B = C_{1,5} \otimes \ldots \otimes C_{n,5} \)
7. \( C = \text{enrich } B \text{ by } (\emptyset, (X, \text{dom}(O)), (\emptyset, \emptyset), (K, I, U, O')) \)

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 \( \text{dom}(O) \cup O \) preserve \( I \). By hypothesis, \( o \text{ preserve } \bigwedge_{1 \leq i \leq n} I_i \), and operations in \( \text{dom}(O) \) 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 preferred, then integrity can be not preserved and soundness is only guaranteed by some non local proofs. In summary, the assembly constructions of B satisfy:

<table>
<thead>
<tr>
<th></th>
<th>regularity</th>
<th>integrity</th>
<th>locality</th>
</tr>
</thead>
<tbody>
<tr>
<td>independent includes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>sharing includes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>using</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

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


