Composition and Refinement in the B-Method

Marie-Laure Potet and Yann Rouzaud
LSR-IMAG, Grenoble, France

Laboratoire Logiciels Systèmes Réseaux - Institut IMAG (UJF - INPG - CNRS)
BP 72, F-38402 Saint Martin d'Hères Cedex - Fax +33 4 76827287
`Marie-Laure.Potet@imag.fr`, `Yann.Rouzaud@imag.fr`

Abstract. In this paper, we propose a framework to study refinement of abstract machines in the B-method. It allows us to properly deal with shared variables, possibly introduced by composition primitives SEES and IMPORTS. We exhibit local conditions on components which are sufficient to ensure global correctness of a software system. Finally, we show how restrictions on the architecture of software systems may guarantee these conditions.

1 Introduction

Modularity is pointed out as a principle allowing to master the complexity of software development or maintenance. A modular method must help designers to produce software systems from autonomous components. Modularity must be offered at each level of software development: programming as well as specification and design.

In the framework of formal methods, *modules* or *components* correspond to syntactic entities which can be combined by *composition primitives*. If a method offers stepwise refinement, adding modularity requires to precisely define composition primitives and how they interact with the process of refinement. Modularity and refinement has been widely studied in the framework of algebraic specifications [ST88]. In the framework of model oriented specifications, some methods offer a concept of modularity (for instance B [Abr96], VDM [Gro94]) but problems appear when combining refinement and composition.

The work presented in this paper was initially motivated by an example communicated by P. Behm, from Matra Transport International. This example exhibits an incorrect B-development in which each component refinement is locally correct. We aimed to extend the architectural conditions given in the B-book (transitivity and circularity, p. 583), in order to detect such pathological examples. So we developped a framework to define refinement of structured components, in order to prove their correctness. This paper presents a simplified version of this framework.

Section 2 presents the B-clauses which allows designers to build structured development (in particular SEES and IMPORTS clauses), and we give a paradigmatic example, illustrating the problem due to refinement composition (section 2.4). In section 3, we propose a semantics for structured components, in terms of *flat components*. From this definition, we show that the proofs obligations relative to each refinement step of structured components, as defined by the B-method, are correct. Then, we introduce a notion of *code component* and we exhibit sufficient conditions to prove its correctness. This amounts to study how monotonicity and transitivity of the refinement relation interact. By a top-down approach we propose several sufficient conditions. The last one is presented in section 4. It is based on the dependency graph between components and it corrects the conditions proposed in the B-Book.

1.1 Composition Primitives

Designing appropriate composition primitives for a specification language or method is a non-trivial task. During the various phases of the software life cycle, expected characteristics can differ. For instance, at the stage of system specification, composition primitives must allow to easily combine pieces of specifications and, if possible, their properties. At the architectural design stage, major system components and their inter-relationships must be identified. So composition primitives must favour the independence of coding activity. This duality is illustrated by the *open-closed* principle [Mey88]. The open view means building larger components by extension. The closed view means making a component available for blind use by others components.

So a method should offer several styles of composition primitives, suitable for the stages of specification, design or programming.

1.2 Composition and Refinement

In a method which offers stepwise refinement, the relationship between different levels of specification is defined by a *refinement relation*. When specifications use composition primitives, a way to refine such structured specifications must be included. At the specification level, some structured specifications can be interpreted as new "flattened" specifications. In this case, the new resulting specification can be refined in any way. Otherwise, if the structure is inherent in the specification, the refinement must preserve the global structure. For instance, in the Z notation [Spi88], a reference to a schema is interpreted as a local copy of the schema text and the new schema may be refined in an independent way. On the contrary, in the programming language Ada, links between packages which are introduced at the specification level, are implicitly preserved in the body.

At the level of architectural design, the final structure of the system is elaborated and components will be implemented separately. This structure must be kept by the latter steps of refinement. This is the property of *compositional refinement* (also referenced as the property of horizontal refinement), which permits to compose refinements to form a large composite refinement architecture.

Refinement composition abilities naturally depend on the relations between components (see for instance discussions about inheritance in the object-oriented approach [LW94]). Several cases can be considered:

1. Refinement composition is always valid. This is the case for instance if the refinement relation is proved to be monotonic [BW90], for such compositions.
2. Global correctness depends on the semantics of the involved components. In this case, a case-by-case proof is mandatory. For instance, if a specification is built as an extension of another specification, the compositionality of refinement depends on the form of this extension.
3. Global correctness depends on the global structure of the system. This is generally the case when sharing is allowed.

Sharing occurs when the same object is used into several modules. In that case, two main problems arise. First, semantics of sharing must be preserved by the refinement process. For instance, if an abstract data type is duplicated, conversion between the different representations must be insured. Secondly, correctness of each independent development does not necessarily guarantee correctness of the composition. Some possible interferences relative to the shared part can break down refinement. This is the classical problem about aliasing and side effects. This point is crucial in the method/language dealing with with states, as we will see.

2 Component, Composition and Refinement in B

2.1 B-Components

In the B-method, there are three kinds of component: *abstract machines*, *refinements* and *implementations*. *Abstract machines* are the visible part of specifications. In particular, all composition primitives connect components with abstract machines. *Refinements* are intermediary steps between interface (abstract machines) and executable code (implementations). These components can be seen as traces of a development. *Implementations* are the last level of a development. Implementations are particular refinements in which substitutions are executable, so they can be translated into code. Moreover, either variables are concrete ones, or they come from other abstract machines, so they must be used via operation calls.

The introduction of several kinds of component, linked to the different steps of the development process, is a particularity of the B-method. In other languages or methods, there is generally only one kind of component, and some syntactic restrictions characterize implementations. This distinction in the B-method is based on the following arguments:

1. Refinements and implementations are not only specifications, but they also contain information about the refinement (the *gluing invariant*, describing the change of variables).
2. Each kind of components has specific syntactic restrictions. For instance, sequencing cannot occur in an abstract machine, in order to favour the abstraction at the first level of a development.
3. The target of composition primitives is always an abstract machine. This restriction favours the decomposition/composition criterion which permits to develop pieces of specification in an independent way.

2.2 B Composition Primitives

The B-method offers, at present, four composition primitives. Two of them (INCLUDES and IMPORTS) exclude possibility of sharing and two of them (USES and SEES) allow a controlled form of sharing.

Includes/Imports. The INCLUDES primitive links abstract machines or refinements to abstract machines. This primitive can be seen as a schema inclusion in Z, without possibility of sharing: this primitive is interpreted as a local copy of the included machines. Due to some syntactic restrictions, the INCLUDES primitive permits to extend an abstract machine (enforcing and adding variables state, adding and hiding operations).

The IMPORTS primitive links implementations to abstract machines. This primitive corresponds to classical component primitives of programming languages. It allows to build a layered software. This primitive can be seen as the closed version of the INCLUDES primitive: use of IMPORTS encapsulates the state of the imported abstract machines.

Implementions are not refinable in the B-method, so the IMPORTS primitive is a final composition primitive. A composition using INCLUDES primitive, with its copy semantics, is not necessarily preserved during the refinement process. If an abstract machine M is included in a component C, being either an abstract machine or a refinement, there are two possibilities:

1. C is refined by its proper structure. In this case, the abstract machine M will be implemented only if it is imported in another implementation.
2. C is implemented using a IMPORTS primitive on M. This possibility is not directly supported by the method, because refinement does not exploit the preservation of this kind of structure.

$$\begin{tikzcd}
C \arrow[r, "inc"'] \arrow[d, Rightarrow] & M \\
I \arrow[ur, "i"'] &
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
C \arrow[r, "inc"'] \arrow[d, Rightarrow] & M \\
I &
\end{tikzcd}$$

Fig. 1. If a component C includes a machine M, an implementation I of C may or may not import M.

Uses/Sees. The USES primitive introduces a form of sharing between abstract machines. In the B-method, sharing is introduced by a link on a component entity. The use of this primitive permits to extend an abstract machine in multiple ways. A final abstract machine must include all shared machines and their extensions. For more explanations about this construction, see [BPR96].

The SEES primitive can appear in abstract machines, refinements or implementations. The use of this primitive allows to share sets, definitions and variables in a very limited way: variables must not be modified by the seing components, in any way. From the refinement point of view, there are some restrictions:

1. abstract machines containing a USES primitive are not considered as refinable specifications.
2. if a SEES primitive is introduced at a given level of a development, this primitive must be preserved in the lowest levels of the development, to guarantee the unicity of the implementation of the shared part.

$$\begin{tikzcd}
C_1 \arrow[r, "s"] \arrow[d, Rightarrow] & M_2 \arrow[d, Rightarrow] \\
I_1 \arrow[ur, "s"'] & I_2
\end{tikzcd}$$

Fig. 2. A SEES primitive must be preserved in a refinement.

The USES and INCLUDES primitives are only syntactic facilities. So in the study of refinement composition, only the SEES and IMPORTS primitives have to be considered.

Comparison between Sees and Imports. The SEES and IMPORTS primitives differ in their use, due to their proper restrictions, whose aim is to limit interference between local refinements. Some restrictions fall on local use of these primitives and some of them are relative to a development, taken as a whole.

Local Restrictions. When an IMPORTS primitive is used, values of variables are accessible only by operation calls. This restriction guarantees the invariant preservation. Moreover, imported variables can occur in the invariant part. This possibility allows designers to constrain imported variables and use them to represent abstract variables. In this way, a layered software is produced. When a SEES primitive is used, variables can be consulted (directly or via operation calls), but cannot be modified. Variables of the seen machine are not visible in the invariant part of the seeing component. As a result, seen variables cannot be used in a refinement to represent abstract variables.

Global Restrictions. Abstract machines which are seen by other ones must be imported once, and only once, in the development. Abstract machines can be imported at most once in a development, so variables cannot be shared by this way (if necessary, renaming, which produces a new copy of a machine, can be used). Another important property is the absence of cycle: an abstract machine cannot see or import itself, directly or indirectly.

2.3 Refinement of a Single B-Component

Refinement relates an "abstract" model of a B-component to a more "concrete" one. In the B-method, it is based on observational substitutivity: any behaviour of the refined specification is one of the possible behaviours of the initial specification. More specifically, B-refinement allows designers to reduce non-determinism of operations, to weaken their preconditions, and to change the variable space.

In the following, we recall some results on B-refinement (chapter 11 of the B-book).

Refinement Component. A refinement component is defined as a differential to be added to a component. A refinement component can have proper variables which are linked to variables of the refined component by a *gluing invariant*. Moreover, refined operations must be stated on the new variables.

```
MACHINE
    $M_1$
VARIABLES
    $v_1$
INVARIANT
    $L_1$
INITIALISATION
    $U_1$
OPERATIONS
    op =
        PRE
            $P_1$
        THEN
            $S_1$
        END
END
```

```
REFINEMENT
    $R_2$
REFINES
    $M_1$
VARIABLES
    $v_2$
INVARIANT
    $L_2$
INITIALISATION
    $U_2$
OPERATIONS
    op =
        PRE
            $P_2$
        THEN
            $S_2$
        END
END
```

```
MACHINE
    $M_2$
VARIABLES
    $v_2$
INVARIANT
    $\exists v_1 .(L_1 \wedge L_2)$
INITIALISATION
    $U_2$
OPERATIONS
    op =
        PRE
            $P_2 \wedge$
            $\exists v_1 .(L_1 \wedge L_2 \wedge P_1)$
        THEN
            $S_2$
        END
END
```

Fig. 3. Refinement R_2 of M_1, seen as an independant machine M_2.

Proof Obligations. The proof obligations for refinement R_2 of Fig. 3 are, provided that there are no common variables (B-book, p. 530):

1. Initialisation: $[U_2]\neg[U_1]\neg L_2$
2. Operation op: $L_1 \wedge L_2 \wedge P_1 \Rightarrow P_2 \wedge [S_2]\neg[S_1]\neg L_2$

In the most general case, there is a chain of refinements $M_1, R_2, \ldots, R_n$ to be considered. The proof obligation for an operation of R_n is, provided that M_1 and its refinements have no common variables:

$$L_1 \wedge L_2 \wedge \ldots \wedge L_n \wedge P_1 \wedge \ldots \wedge P_{n-1} \Rightarrow P_n \wedge [S_n]\neg[S_{n-1}]\neg L_n \ .$$

2.4 Compositional Refinement

In languages based on states, a major difficulty is relative to the sharing of states. In presence of sharing, we must prove that some local reasoning about values of variables are always valid in a global system. Such a problem appears in the B refinement process.

Example 1. Let A, B, C be the following abstract machines:

```
MACHINE
    A
VARIABLES
    xa
INVARIANT
    xa : 0..1
INITIALISATION
    xa := 0
OPERATIONS
    rr ← val_xa = rr := xa ;
    mod_xa = xa := 1 - xa
END
```

```
MACHINE
    B
OPERATIONS
    opb =
        skip
END
```

```
MACHINE
    C
OPERATIONS
    rr ← opc =
        rr := TRUE
END
```

Now, let CI be the following implementation of C:

```
IMPLEMENTATION CI  REFINES C  IMPORTS B  SEES A
OPERATIONS
    rr ← opc =
        VAR v1, v2 IN
            v1 ← val_xa;  opb;  v2 ← val_xa;  rr := bool(v1=v2)
        END
END
```

This refinement is valid. Using B-definitions on substitutions, we have to prove that TRUE=bool(xa=xa), which is true. Now machine B is implemented, with the help of D and DI, by BI:

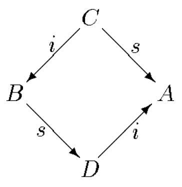
```
IMPLEMENTATION       MACHINE            IMPLEMENTATION
    BI                   D                  DI
REFINES              OPERATIONS         REFINES
    B                    opd = skip         D
SEES                 END                IMPORTS
    D                                       A
OPERATIONS                              OPERATIONS
    opb = opd                               opd = mod_xa
END                                     END
```

These two refinements are also valid. But, despite the fact that proof obligations proposed by the B-method can be discharged, the code of the operation *opc* is not correct (see below).

```
rr ← opc =
    VAR v1, v2 IN
        v1 := xa ;  xa := 1 - xa ;  v2 := xa ;  rr := bool(v1=v2)
    END
```

The resulting substitution is bool(xa=(1-xa)), which is FALSE. Where is the flaw? When implementing the abstract machine C, we implicitly suppose that the operation *opd* of the machine D does not affect the variable xa. But this hypothesis is broken by the implementation *DI* (see Fig. 4).

The B-method imposes conditions on architecture to eliminate some incorrect cases (B-book, p. 583): it is not possible for a machine, a refinement or an implementation to see a machine that is one of its ancestors or descendants through a chain of IMPORTS primitives. But the architecture of our example does not fit this condition, because A is imported through a SEES primitive.



$$\begin{tikzcd} & C \arrow[dl, "i"'] \arrow[dr, "s"] & \\ B \arrow[dr, "s"'] & & A \\ & D \arrow[ur, "i"'] & \end{tikzcd}$$

Fig. 4. Architecture of Example 1.

The problem comes from two different views on abstract machines. When abstract machines are combined, only modifications described in the abstract definition of operations are observable. So, we implicitly suppose that abstract machines, and *a fortiori* their code, do not modify anything else. When abstract machines are refined, new variables can be introduced. So we implicitly suppose that operations can alter other variables, in a way compatible with the refinement invariant.

If variables introduced by a refinement are local to this refinement, the composition is valid. But if these variables can be referenced in other abstract machines by composition, these two views can become inconsistent and some unpleasant side effects can appear. New conditions are necessary to simultaneously adopt these two points of view.

2.5 Notation, Operations and Properties on Refinements

In this paper, in order to highlight the essential part of a proof obligation, the notation $\sqsubseteq_L$ will be used, and gluing invariants of intermediate refinements, as well as preconditions of their operations, will be omitted and considered as hidden hypotheses. We will also assume that the precondition of the refining operation is already proved. So, *in such an implicit context*, the proof obligation of an operation op, refined in a chain of refinements $R_1, \ldots, R_n$, will be written $L_n \Rightarrow op_{R_{n-1}} \sqsubseteq_{L_n} op_{R_n}$, where L_n is the gluing invariant of R_n.

Definition 1. *Refinement Relation* $\sqsubseteq_L$.

Let L be a predicate, $op_1 \equiv P_1|S_1$ and $op_2 \equiv P_2|S_2$ be two operations with the same signature. Then:

$$op_1 \sqsubseteq_L op_2 \equiv [S_2]\neg[S_1]\neg L \ .$$

Definition 2. *Notation var and free.*

1. $var(C)$ is the set of variables of the component C, in the VARIABLES clause.
2. $free(L)$ is the set of free variables of the predicate L.

Renaming Common Variables. When a B-component C and its immediate refinement R, with gluing invariant L, share some variables v, a renaming must be introduced, in order to properly deal with proof obligations. Let v' be a set of fresh variables, related to v. Then v will be renamed by v' in R (and in the chain of refinements beginning with R), so the proof obligation for an operation op becomes:

$$L \wedge v = v' \Rightarrow op_C \sqsubseteq_{L \wedge v = v'} [v := v']op_R \ .$$

Translating a B-Refinement into an Independant Machine. This operation takes a refinement R_n in a chain $M_1, R_2, \ldots, R_n$, and delivers the corresponding independant abstract machine M_n, which looks like abstract machine M_2 of Fig. 3. Main characteristic of this translation is that intermediate variables are hidden by existential quantification. Notice that renaming of common variables is prerequisite. Invariant of the resulting machine is:

$$\exists x_1, \ldots, x_{n-1} \cdot (L_1 \wedge L_2 \wedge \ldots \wedge L_n)$$

and the precondition of an operation is:

$$P_n \wedge \exists x_1, \ldots, x_{n-1} \cdot (L_1 \wedge L_2 \wedge \ldots \wedge L_n \wedge P_1 \wedge \ldots \wedge P_{n-1}) \ .$$

Reducing a Chain of Refinements. Reducing a chain of refinements M_1, $R_2, \ldots, R_n$ consists in defining a direct refinement R'_n of abstract machine M_1. Let M_n be the independant abstract machine, corresponding with R_n. Then R'_n is the differential to be added to M_1, in order to build M_n. Notice that renaming of common variables is prerequisite. Gluing invariant between M_1 and R'_n is:

$$\exists x_2, \ldots, x_{n-1} \cdot (L_2 \wedge \ldots \wedge L_n)$$

and the precondition of an operation of R'_n is:

$$P_n \wedge \exists x_1, \ldots, x_{n-1} \cdot (L_1 \wedge \ldots \wedge L_n \wedge P_1 \wedge \ldots \wedge P_{n-1}) \ .$$

Invariant Splitting. In the following, the invariant splitting property will be used to establish sufficient conditions for a proof obligation of a refinement, when its gluing invariant L takes the form $L_1 \wedge L_2$.

Lemma 1. Let S be a substitution and P, Q be two predicates, such that S does not modify the free variables of Q. We have:

$$Q \wedge \neg[S]\neg P \Rightarrow \neg[S]\neg(P \wedge Q).$$

Proof: by structural induction on substitutions.

Lemma 2. Let S_1 and S_2 be two substitutions, and A, B two predicates, such that S_1 does not modify the free variables of B. We have:

$$[S_1]B \wedge [S_2]\neg[S_1]\neg A \Rightarrow [S_2]\neg[S_1]\neg(A \wedge B)$$

Notice that in general, we cannot deduce $[S_2]\neg[S_1]\neg(A \wedge B)$ from the hypotheses $[S_2]\neg[S_1]\neg A$ and $[S_2]\neg[S_1]\neg B$.

Proof: by lemma 1, we have $B \wedge \neg[S_1]\neg A \Rightarrow \neg[S_1]\neg(A \wedge B)$. By monotonicity of substitutions through implication (B-Book, p. 287), we obtain $[S_2](B \wedge \neg[S_1]\neg A) \Rightarrow [S_2]\neg[S_1]\neg(A \wedge B)$. The property is then established by distributivity of substitutions through conjunction (B-Book, p. 287).

Property 1. *Invariant splitting.*

Let C be a B-component, R a refinement of C, $L \equiv L_1 \wedge L_2$ the gluing invariant of R, and op_C an operation of C, whose refinement is op_R. Property $op_C \sqsubseteq_L op_R$ holds if:

1. op_C does not modify the free variables of L_2,
2. $[op_R]L_2$,
3. $op_C \sqsubseteq_{L_1} op_R$.

Proof: direct application of lemma 2.

3 A framework for Compositional Refinement

We call *B-component* a B-entity: an abstract machine, a refinement or an implementation. A B-component is *flat* if it includes neither SEES nor IMPORTS primitive. Otherwise it is *structured.*

First, we propose a semantics for structured components. Following the work presented in [BPR96], the chosen semantics consists in interpreting such components as new "flattened" components. Thus refinement of structured components can be reduced to refinement of flat components. Finally, we use this framework to define the last step of a development: how the code of an abstract machine is elaborated. Studying the correctness of this code comes down to study the monotonicity of the refinement relation with respect to the structural SEES and IMPORTS primitives. This form of monotonicity is not always valid (recall example 1), and some sufficient conditions will be pointed out.

3.1 Flattening Structured B-Components

We define a *flattening* operation, denoted by $\mathcal{F}$, which produces a new flat component, in which all SEES and IMPORTS primitives are expanded. In such a flat component, the keywords MACHINE, REFINEMENT, IMPLEMENTATION are replaced with COMPONENT. If no REFINES clause appears in a component, it comes from an abstract machine. Otherwise, it comes from a refinement or an implementation. This change of keyword is done to underline that there is no syntactic restriction on the allowed substitutions in our components.

In the flattening operation, we only consider variables and clauses related to variables (initialisation, invariant and operations), because problems of refinement composition come from variables. The SEES and IMPORTS primitives will be treated in the same way, because they have the same semantics (call of operations on an encapsulated state). The difference lays on the possibility of sharing for the SEES primitive: in this case some components $\mathcal{F}(M_i)$ can have some variables in common, coming from seen machines.

Definition 3. *The Flattening Operation.*

Let C be a B-component. If C is stand-alone, then $\mathcal{F}(C)$ is C, with the header "COMPONENT $\mathcal{F}(C)$". Otherwise, C has some SEES or IMPORTS primitives on machines $M_1, \ldots, M_n$. The flat component $\mathcal{F}(C)$ is defined as follows:

1. Header of $\mathcal{F}(C)$ is "COMPONENT $\mathcal{F}(C)$".
2. If C refines a B-component C', then a clause "REFINES $\mathcal{F}(C')$" is introduced.
3. Variables of $\mathcal{F}(C)$ are variables of C, $\mathcal{F}(M_1), \ldots, \mathcal{F}(M_n)$. Because variables of C, $M_1, \ldots, M_n$ are distinct (a restriction imposed by the B-method), common variables may only come from the machines which are seen (several times) and imported (almost once).
4. Invariant of $\mathcal{F}(C)$ is the conjunction of invariant of C and invariants of $\mathcal{F}(M_1), \ldots, \mathcal{F}(M_n)$. For the same reason as above, invariants on shared variables are necessarily identical.

5. Initialisation of $\mathcal{F}(C)$ is the substitution $((U_1 \otimes \ldots \otimes U_n)\ ; U)$, where each U_i is the initialisation of the component $\mathcal{F}(M_i)$ and U is the initialisation of M. The operator $\otimes$ is the extension of the operator $||$ when variables are shared (see [BPR96] for more explanations).
6. Operations of $\mathcal{F}(C)$ are expanded operations of C. Expansion consists in replacing the calls to operations with their bodies, where formal parameters are replace with effective parameters (B-book, page 314). We suppose here that operations are not recursive.

Property 2. *Invariant Preservation by an Operation Call.*

Let M be a component corresponding abstract machine and I be its invariant. It can be proved that I is preserved by a substitution S, calling operations of M, if these operations are called into their precondition. Such a condition is imposed by the B-method. In consequence, for a component C, seeing or importing a component M, each operation of $\mathcal{F}(C)$ preserves the invariant of M.

Example 2. The flat component associated with implementation DI of Example 1 is:

```
COMPONENT $\mathcal{F}(DI)$  REFINES $\mathcal{F}(D)$
VARIABLES xa
INVARIANT xa : 0..1
INITIALISATION xa := 0
OPERATIONS opd =  xa := 1 - xa
END
```

3.2 Structured Refinement

Let C be a B-component, seeing abstract machines $M_1, \ldots, M_k$, and let R be a B-refinement of C, seeing the same machines, and possibly seeing or importing other machines $M_{k+1}, \ldots, M_n$. We suppose here that common variables between $\mathcal{F}(C)$ and $\mathcal{F}(R)$ only come from seen machines, i.e. $M_1, \ldots, M_k$ (other common variables can be renamed, if necessary).

To prove the correctness of this refinement, we have to prove that $\mathcal{F}(C)$ is refined by $\mathcal{F}(R)$:

1. By the flattening operation, invariant of $\mathcal{F}(R)$ is $L \wedge L_1 \ldots \wedge L_n$, where L is the gluing invariant between C and R and each L_i is the invariant of $\mathcal{F}(M_i)$.
2. Because $\mathcal{F}(C)$ and $\mathcal{F}(R)$ have some common variables (variables of $M_1, \ldots, M_k$), renaming must be done and the gluing invariant must be strengthened. Let v_s be this set of variables and v'_s be a set of corresponding fresh variables. We rename v_s by v'_s in the component $\mathcal{F}(R)$ and the new invariant becomes $L \wedge L_1 \ldots \wedge L_n \wedge v_s = v'_s$. Thus we must establish, for each operation of C:

$$L \wedge L_1 \ldots \wedge L_n \wedge v_s = v'_s \Rightarrow op_{\mathcal{F}(C)} \sqsubseteq_{L \wedge L_1 \ldots \wedge L_n \wedge v_s = v'_s} [v_s := v'_s] op_{\mathcal{F}(R)}$$

3. Two applications of the splitting invariant property will simplify this formula:
   (a) Splitting into $L_1 \wedge \ldots \wedge L_k \wedge v_s = v'_s$ and $L \wedge L_{k+1} \wedge \ldots \wedge L_n$.
      i. $op_{\mathcal{F}(C)}$ does not modify variables of $L_1 \wedge \ldots \wedge L_k \wedge v_s = v'_s$: variables v'_s are fresh variables, and, for variables v_s, only consulting operations can be called in $op_{\mathcal{F}(C)}$.
      ii. With similar arguments about $op_{\mathcal{F}(R)}$, $[[v_s := v'_s]op_{\mathcal{F}(R)}](L_1 \wedge \ldots \wedge L_k \wedge v_s = v'_s)$ can be reduced to $L_1 \wedge \ldots \wedge L_k \wedge v_s = v'_s$, which belongs to hypotheses.
      iii. So it remains to prove $L \wedge L_1 \wedge \ldots \wedge L_n \wedge v_s = v'_s \Rightarrow$ $op_{\mathcal{F}(C)} \sqsubseteq_{L \wedge L_{k+1} \wedge \ldots \wedge L_n} [v_s := v'_s]op_{\mathcal{F}(R)}$, which is equivalent to $L \wedge L_1 \wedge \ldots \wedge L_n \Rightarrow op_{\mathcal{F}(C)} \sqsubseteq_{L \wedge L_{k+1} \wedge \ldots \wedge L_n} op_{\mathcal{F}(R)}$ (proof by structural induction on substitutions).
   (b) Splitting into L and $L_{k+1} \wedge \ldots \wedge L_n$.
      i. The operations $op_{\mathcal{F}(C)}$ do not modify variables of $L_{k+1}, \ldots, L_n$ because variables of these machines are not accessible from C.
      ii. By property 2, $L \wedge L_1 \wedge \ldots \wedge L_n \Rightarrow [op_{\mathcal{F}(R)}]L_i$ holds for each i.
      iii. So it remains to prove $L \wedge L_1 \wedge \ldots \wedge L_n \Rightarrow op_{\mathcal{F}(C)} \sqsubseteq_L op_{\mathcal{F}(R)}$.

In conclusion, the final condition is $L \wedge L_1 \wedge \ldots \wedge L_n \Rightarrow op_{\mathcal{F}(C)} \sqsubseteq_L op_{\mathcal{F}(R)}$, which is the one proposed by Atelier-B [AtB] in presence of SEES or IMPORTS primitives.

3.3 Code Components

In this section, we introduce the notion of code component, in order to build the code attached to abstract machines. Code components are flat components in which references to abstract machines, introduced by SEES or IMPORTS clauses, are replaced by the code associated with these abstract machines. In the following, we define two kinds of code component:

1. $\mathcal{C}(I)$ is the code component refining $\mathcal{F}(I)$, if I is an implementation.
2. $\mathcal{C}(M)$ is the code component refining $\mathcal{F}(M)$, if M is an abstract machine. It is obtained from $\mathcal{C}(I)$ by reducing the refinement chain $\mathcal{F}(M)$, $\mathcal{F}(I)$, $\mathcal{C}(I)$.

For simplicity reasons, we suppose that variables of an implementation can only come from seen and imported machines (dealing with local concrete variables should not be a problem).

Definition 4. *Code Component Operation $\mathcal{C}$.*

1. Let I be a B-implementation and let I' be I, without its gluing invariant and with the clause "REFINES $\mathcal{F}(I)$".
   (a) If I has neither SEES nor IMPORTS primitive, I has no variables (see above), and $\mathcal{C}(I)$ is I'.

   (b) If I is a structured B-implementation with SEES or IMPORTS primitives on components $M_1, \ldots, M_n$, $\mathcal{C}(I)$ is obtained by flattening together I' and the code components $\mathcal{C}(M_1), \ldots \mathcal{C}(M_n)$. The resulting invariant of $\mathcal{C}(I)$ is $L_1 \wedge \ldots \wedge L_n$, where each L_i is the invariant of $\mathcal{C}(M_i)$.
2. Let M be a B-abstract machine.
   (a) If M is a basic machine, $\mathcal{C}(M)$ is obtained from $\mathcal{F}(M)$ by adding the clause "REFINES $\mathcal{F}(M)$", by renaming its variables v with fresh variables v', then by adding to its invariant the gluing invariant $v = v'$. Recall that a basic machine has no B-implementation.
   (b) If M has the implementation I, $\mathcal{C}(M)$ is obtained by reducing the refinement chain $\mathcal{F}(M)$, $\mathcal{F}(I)$, $\mathcal{C}(I)$, as defined in section 2.5.

Property 3. *Code Component Variables.*

1. Let C be a B-component, then variables of its code $\mathcal{C}(C)$ only come from the code of basic machines: $var(\mathcal{C}(C)) \subseteq \bigcup\{var(\mathcal{C}(M)) : M \text{ is a basic machine}\}$.
2. Let I a B-implementation. Since variables of the code of basic machines are fresh variables, $var(\mathcal{F}(I)) \cap var(\mathcal{C}(I)) = \emptyset$.

Property 4. *Variables of Gluing Invariants of Code Components.* Let L be the gluing invariant of a code component $\mathcal{C}(M)$, where M is an abstract machine; we have: $free(L) = var(\mathcal{F}(M)) \cup var(\mathcal{C}(M))$.

Example 3. We suppose here that the abstract machine A of example 1 is a basic machine. So the code components $\mathcal{C}(D)$ and $\mathcal{C}(DI)$, respectively associated with components $\mathcal{F}(D)$ and $\mathcal{F}(DI)$ are:

```
COMPONENT
    𝒞(D)
REFINES
    ℱ(D)
VARIABLES
    xa'
INVARIANT
    ∃ xa . (xa : 0..1 ∧ xa' : 0..1 ∧ xa = xa')
INITIALISATION
    xa' := 0
OPERATIONS
    opd =
        xa' := 1 - xa'
END
```

```
COMPONENT
    𝒞(DI)
REFINES
    ℱ(DI)
VARIABLES
    xa'
INVARIANT
    xa' : 0..1 ∧ xa = xa'
INITIALISATION
    xa' := 0
OPERATIONS
    opd =
        xa' := 1 - xa'
END
```

3.4 Code Correctness

Now we have to prove that $\mathcal{F}(I)$ is refined by $\mathcal{C}(I)$. If I has neither SEES nor IMPORTS primitive, the proof is obvious. Otherwise, the condition is:

Condition 1. *A Compositional Proof Obligation.*
If $\mathcal{F}(I)$ has been obtained from a structured B-implementation with some SEES or IMPORTS primitives on components $M_1, \ldots, M_n$, it suffices to prove :

$$\boxed{L_1 \wedge \ldots \wedge L_n \Rightarrow op_{\mathcal{F}(I)} \sqsubseteq_{L_1 \wedge \ldots \wedge L_n} op_{\mathcal{C}(I)}}$$

where each L_i denotes the invariant of $\mathcal{C}(M_i)$, i.e. the gluing invariant between $\mathcal{C}(M_i)$ and $\mathcal{F}(M_i)$.

3.5 A Sufficient Condition

Condition 1 cannot be directly reduced using the splitting invariant property, so we now inspect the structure of operations.

This analysis only works when SEES primitives only occur at the level of implementations. In this case, we have $var(\mathcal{F}(M)) = var(M)$, for any machine M. A complete analysis, giving the same results, will be published later.

1. Because operations in $\mathcal{F}(I)$ and $\mathcal{C}(I)$ only differ in the expansion of the operations which are called in I, the property of monotonicity of refinement can be used (B-Book, p. 504). Thus, operations of $\mathcal{C}(I)$ refine operations of $\mathcal{F}(I)$ if we can prove that, for each i, $L_1 \wedge \ldots \wedge L_n \Rightarrow op_{\mathcal{F}(M_i)} \sqsubseteq_{L_1 \wedge \ldots \wedge L_n} op_{\mathcal{C}(M_i)}$. This use of monotonicity amounts to prove that gluing invariants L_j are also verified by operations op_{M_i} and their refinements.
2. Now the invariant splitting property can be used:
   (a) Operations $op_{\mathcal{F}(M_i)}$ cannot modify variables in $free(L_j)$ for $i \neq j$, because, by property 4, $free(L_j) = var(\mathcal{F}(M_j)) \cup var(\mathcal{C}(M_j))$: variables of abstract machines are supposed to be disjoint (after renaming if necessary), variables of a code are fresh variables, and op_{M_i} can only call consulting operations.
   (b) $L_1 \wedge \ldots \wedge L_n \Rightarrow op_{\mathcal{F}(M_i)} \sqsubseteq_{L_i} op_{\mathcal{C}(M_i)}$ is a consequence of the refinement proof obligation on M_i, which is $L_i \Rightarrow op_{\mathcal{F}(M_i)} \sqsubseteq_{L_i} op_{\mathcal{C}(M_i)}$.
   (c) Then it suffices to prove $L_1 \wedge \ldots \wedge L_n \Rightarrow [op_{\mathcal{C}(M_i)}](\bigwedge_{j \neq i} L_j)$.
3. Using distributivity of substitution through conjonction, we obtain the following sufficient condition:

Condition 2. *A sufficient condition.*
If $\mathcal{F}(I)$ has been obtained from a structured B-implementation with some SEES or IMPORTS primitives on components $M_1, \ldots, M_n$, a sufficient condition is, for each i and j with $i \neq j$:

$$\boxed{L_1 \wedge \ldots \wedge L_n \Rightarrow [op_{\mathcal{C}(M_i)}]L_j}$$

where each L_i denotes the invariant of $\mathcal{C}(M_i)$, i.e. the gluing invariant between $\mathcal{C}(M_i)$ and $\mathcal{F}(M_i)$.

4 An Architectural Condition

The sufficient condition stated above preserves, in some sense, the composition of refinement because proof obligations of each local refinement are reused. But new proofs are necessary. Less fine sufficient conditions can be stated on the architecture of developments, in order to guarantee that no potentially incorrect configuration appears. For that purpose, first we define some dependency relations between abstract machines. Secondly a finer analysis of gluing invariants of code components is proposed, using a restriction on the SEES primitive. Finally, we examine the sufficient condition in terms of easily checkable conditions on dependencies.

Definition 5. *Dependency relations.*

1. M_1 *sees* M_2 iff the implementation of M_1 sees the machine M_2.
2. M_1 *imports* M_2 iff the implementation of M_1 imports the machine M_2.
3. M_1 *depends_on* M_2 iff the code of M_1 is built by using M_2: $\mathit{depends\_on} = (\mathit{sees} \cup \mathit{imports})^+$.
4. M_1 *can_consult* M_2 iff the code of M_1 can consult the variables of the code of M_2: $\mathit{can\_consult} = (\mathit{depends\_on}^*; \mathit{sees})$.
5. M_1 *can_alter* M_2 iff the code of M_1 can modify the variables of the code of M_2: $\mathit{can\_alter} = (\mathit{depends\_on}^*; \mathit{imports})$.

Relational notation is the one of the B-method: transitive closure ($^+$), reflexive and transitive closure (*) and composition (;).

4.1 Variables and Dependency Relations

To ensure the sufficient condition $[op_{\mathcal{C}(M_i)}]L_j$ in terms of dependency relations, a condition is the following: variables which both appear in $op_{\mathcal{C}(M_i)}$ and in L_j cannot be modified by $op_{\mathcal{C}(M_i)}$. To state this condition, $var(C)$, the set of variables of a component C must be analyzed, in the case of a code component.

Property 5. *Variables of Code Components.*

1. For a basic machine M, $var(\mathcal{C}(M))$ is the set of variables obtained from $var(M)$ by renaming variables v by v'.
2. For a non-basic machine M, $var(\mathcal{C}(M))$ is the set of the variables of the code of an abstract machine which is in the dependency graph of $\mathcal{C}(M)$, i.e.: $var(\mathcal{C}(M)) = \bigcup\{var(\mathcal{C}(N)) : N \in \mathit{depends\_on}[\{M\}]\}$.

Now we come back to the sufficient condition. Variables of $op_{\mathcal{C}(M_i)}$ which can be modified come from code of machines in the set $\mathit{can\_alter}[\{M_i\}]$. On the other hand, by properties 4 and 5, free variables of L_j come from machines or their code in the set $\{M_j\} \cup \mathit{depends\_on}[\{M_j\}]$. Because $\{M_j\} \subseteq \mathit{depends\_on} * [\{M_j\}]$, condition 2 is ensured if for each i and j, with $j \neq i$:

$$\mathit{can\_alter}[\{M_i\}] \cap \mathit{depends\_on}^*[\{M_j\}] = \emptyset \ .$$

This structural condition is too restrictive, because it rejects too many architectures. For instance, architecture of Fig. 5 does not fit this condition but can be proved correct if refinements are locally correct:

$$\begin{tikzcd}
 & M_1 \arrow[dl, "i"'] \arrow[dr, "i"] & \\
M_2 \arrow[dr, "s"'] & & M_3 \arrow[dl, "i"] \\
 & M_4 &
\end{tikzcd}$$

Fig. 5. A correct architecture.

4.2 Using Restrictions on Clauses

In this section, a finer analysis of the gluing invariant of code components is made, using a restriction specific to the SEES primitive: variables coming from a seen machine cannot be referenced into gluing invariants of seing components. In consequence, a continuous chain of IMPORTS primitives is needed to alter variables of an abstract machine: it explains why architecture of Fig. 5 is correct.

Property 6. *Form of the Gluing Invariant of Code Components.*
Let L be the gluing invariant of $\mathcal{C}(M)$, where M is an abstract machine with no SEES primitive, then L takes the form $A \wedge B$, with:

1. $\mathit{free}(A) = \bigcup\{\mathit{var}(\mathcal{C}(N_i)) : N_i \in \mathit{can\_consult}[\{M\}]\}$. In this case, $A \equiv \bigwedge A_i$, where each A_i is the invariant of the independant abstract machine corresponding to $\mathcal{C}(N_i)$ (section 2.5).
2. If M is a basic machine $\mathit{free}(B) = \mathit{var}(M) \cup \mathit{var}(\mathcal{C}(M))$.
3. Otherwise $\mathit{free}(B) = \mathit{var}(\mathcal{F}(M)) \cup \bigcup\{\mathit{var}(\mathcal{C}(N)) : N \in \mathit{imports}^+[\{M\}]\}$.

Proof by induction. In the case of a basic machine M, A is true and B is the gluing invariant of $\mathcal{C}(M)$.

Now we analyse the inductive step on a simplified case (with no loss of generality). Let M be an abstract machine and I be its implementation, seeing a machine M_s and importing a machine M_i. Then we have:

1. Invariant of $\mathcal{C}(M_\mathrm{s})$ takes the form $A_\mathrm{s} \wedge B_\mathrm{s}$.
2. Invariant of $\mathcal{C}(M_\mathrm{i})$ takes the form $A_\mathrm{i} \wedge B_\mathrm{i}$.
3. A_s and A_i are conjuctions of invariants of independant machines.
4. $\mathit{imports}^+[\{M\}] = \{M_\mathrm{i}\} \cup \mathit{imports}^+[\{M_\mathrm{i}\}]$.
5. $\mathit{can\_consult}^+[\{M\}] = \{M_\mathrm{s}\} \cup \mathit{can\_consult}^+[\{M_\mathrm{i}\}] \cup \mathit{can\_consult}^+[\{M_\mathrm{s}\}]$.

6. Invariant of $\mathcal{C}(M)$, as defined in section 3.3, is:

$$\exists v_{\mathrm{s}}, v_{\mathrm{i}} \cdot (L \wedge A_{\mathrm{s}} \wedge B_{\mathrm{s}} \wedge A_{\mathrm{i}} \wedge B_{\mathrm{i}} \wedge L_{\mathrm{s}} \wedge L_{\mathrm{i}})$$

   where L is the gluing invariant between $\mathcal{F}(I)$ and $\mathcal{F}(M)$, L_{s} is the invariant of M_{s}, and L_{i} is the invariant of M_{i}.
7. (a) $v_{\mathrm{s}} \notin \mathit{free}(L)$ because a seen variable cannot occur into gluing invariants.
   (b) $v_{\mathrm{s}} \notin \mathit{free}(A_{\mathrm{i}} \wedge B_{\mathrm{i}})$ and $v_{\mathrm{i}} \notin \mathit{free}(A_{\mathrm{s}} \wedge B_{\mathrm{s}})$, thanks to property 5.
   (c) $v_{\mathrm{s}} \notin \mathit{free}(L_{\mathrm{i}})$ and $v_{\mathrm{i}} \notin \mathit{free}(L_{\mathrm{s}})$ because machines have disjoint variables (after renaming if necessary).
   (d) $v_{\mathrm{s}} \notin \mathit{free}(A_{\mathrm{s}})$ and $v_{\mathrm{i}} \notin \mathit{free}(A_{\mathrm{i}})$, by inductive hypothesis.
   (e) So invariant of $\mathcal{C}(M)$ becomes, after putting some subformulae out of the scope of the quantifiers:

$$\exists v_{\mathrm{s}} \cdot (A_{\mathrm{s}} \wedge B_{\mathrm{s}} \wedge L_{\mathrm{s}}) \wedge A_{\mathrm{i}} \wedge \exists v_{\mathrm{i}} \cdot (L \wedge B_{\mathrm{i}} \wedge L_{\mathrm{i}}) \ .$$

8. $A \equiv \exists v_{\mathrm{s}} \cdot (A_{\mathrm{s}} \wedge B_{\mathrm{s}} \wedge L_{\mathrm{s}}) \wedge A_{\mathrm{i}}$ and $\exists v_{\mathrm{s}} \cdot (A_{\mathrm{s}} \wedge B_{\mathrm{s}} \wedge L_{\mathrm{s}})$ is the invariant of the independant machine corresponding to $\mathcal{C}(M_{\mathrm{s}})$.
9. $B \equiv \exists v_{\mathrm{i}} \cdot (L \wedge B_{\mathrm{i}} \wedge L_{\mathrm{i}})$.

4.3 An Architectural Sufficient Condition

Recall that we want to ensure condition $[op_{\mathcal{C}(M_i)}]L_j$ for each i and j, with $i \neq j$, where $M_1, \ldots, M_n$ are seen or imported in the implementation I of machine M, and $L_1, \ldots, L_n$ are respectively the gluing invariants of $[op_{\mathcal{C}(M_1)}], \ldots, [op_{\mathcal{C}(M_n)}]$. We suppose that M has no SEES primitive.

1. Using property 6 and distributivity of substitution through conjonction, L_j takes the form $A_j \wedge B_j$ and condition 2 becomes:
   (a) $L_1 \ldots \wedge L_n \Rightarrow [op_{\mathcal{C}(M_i)}]A_j$
   (b) $L_1 \ldots \wedge L_n \Rightarrow [op_{\mathcal{C}(M_i)}]B_j$
   First formula holds, due to property 2 and to the fact that A_j is a conjonction of invariants of independant machines. So a sufficient condition is second formula.
2. By property 6, free variables of B_j come from machines or their code in the set $\{M_j\} \cup \mathit{imports}^+[\{M_j\}]$. So it suffices to prove that $op_{\mathcal{C}(M_i)}$ cannot modify variables of B_j:

$$\mathit{can\_alter}[\{M_i\}] \cap (\{M_j\} \cup \mathit{imports}^+[\{M_j\}]) = \emptyset \ .$$

3. Using the fact that an abstract machine is imported once only, we obtain:

$$\mathit{can\_alter}[\{M_i\}] \cap \{M_j\} = \emptyset \ .$$

4. Next step consists in stating this condition in terms of machine M.
   (a) If M_j is imported by I then M_i cannot import M_j, so condition holds.

(b) The remaining case is when M_i is seen by I. So condition becomes:

$$can\_alter[\{M\}] \cap sees[\{M\}] = \emptyset \ .$$

5. Now, considering the global architecture of developments, we obtain:

Condition 3. *An Architectural Condition.*
An architecture of developpements, where SEES primitives only occur at the level of implementations, is correct if all components are proved and if:

$$\boxed{can\_alter \cap sees = \emptyset}$$

The B-Book (p. 583) and the Atelier-B (up to version 3.2) propose architectural conditions which can be translated, in terms of our relations, into:

$$(imports^{+} \cup (sees; imports^{+})) \cap sees = \emptyset \ .$$

This condition is not sufficient, because it does not consider *can_alter*. So the incorrect architecture of example 1, which does not respect condition 3, is accepted by the B-method.

5 Conclusion

A practical issue of our work results in a set of conditions to guarantee the correctness of refinements in the presence of SEES and IMPORTS primitives, when SEES primitives only occur at the level of implementations

1. Translation of SEES and IMPORTS primitives in terms of flat components has given condition 1 which consists in proving that refinements can be combined.
2. Use of monotonicity has given stronger condition 2, which exploits the fact that a SEES primitive only allows calls of consulting operations. This condition is simpler to verify than condition 1.
3. Proper restrictions on the clauses SEES and IMPORTS, which can be seen as the impossibility to represent two independent abstract states on the same implementation, have given the final condition 3 on the dependency graph.

A complete analysis, with no restriction on SEES primitive, is under development and it will be published later. Under reasonable assumptions about chains of SEES primitives, it gives the same sufficient conditions.

Under this analysis, it is possible to consider several levels of checkings. For instance, if condition 3 is not valid, we can try to verify condition 2 or 1, before reconsidering the global development, in order to eliminate the undesirable sharing.

The second issue is, following [BPR96], a framework for dealing with structured components in the B-method, in order to study how proofs of invariant properties and proofs of refinement can be combined. Within this framework, extensions of B-method primitives can be proposed. Structural clauses offered by the B-method have some restricting conditions which could be removed. If semantics of new structural clauses can be defined in terms of flat components, then a proper analysis of their properties of compositional refinement can be done in a rigourous way. For instance, the compositionality of some forms of INCLUDES or USES can be studied, resulting in a more flexible form of sharing. Following [MQR95], it seems to us that sharing must be introduced at the specification level and must be strictly controlled in the refinement process. In that case, fine reasoning about sharing is possible. The problem met in the B-method comes from the fact that reasoning is done at the level of abstract machines, in which sharing is not always specified.

References

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

[AtB] Steria Méditerranée, *Le Langage B. Manuel de référence version 1.5* , S.A.V. Steria, BP 16000, 13791 Aix-en-Provence cedex 3, France.

[BPR96] D. Bert, M-L. Potet, Y. Rouzaud, *A Study on Components and Assembly Primitives in B*, In First Conference on the B Method, 47–62, H. Habrias, editor, 1996.

[BW90] R. J. R. Back, J. von Wright, *Refinement calculus I: Sequential Nondeterministic Programs*, In Stepwise Refinement of Distributed Systems, J. W. deBakker, W. P. deRoever, and G. Rozenberg, editors, LNCS 430, 42–66, Springer-Verlag, 1990.

[Gro94] The VDM-SL Tool Group, *User's Manual for the IFAD VDM-SL Toolbox*, IFAD, Forskerparken 10, 5230 Odense M, Denmark, 1994.

[LW94] B. Liskov, J. Wing, *A Behavioural Notion of Subtyping*, ACM Transactions on Programming Languages and Systems, 16(6), 1811–1841, 1994.

[Mey88] B. Meyer, *Object-Oriented Construction*, Prentice-Hall, 1988.

[MQR95] M. Moriconi, X. Qian, R. A. Riemenschneider, *Correct Architecture Refinement*, IEEE Transactions on Software engineering, 21(4), 356–372, 1995.

[Spi88] M. Spivey, *Understanding Z: a Specification Language and its Formal Semantics*, Cambridge University Press, 1988.

[ST88] D. Sannella, 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.