Interpreting Invariant Composition in the B Method Using the Spec# Ownership Relation: A Way to Explain and Relax B Restrictions

Sylvain Boulmé and Marie-Laure Potet

LSR-IMAG, Grenoble, France
Sylvain.Boulme@imag.fr,
Marie-Laure.Potet@imag.fr

Abstract. In the B method, the invariant of a component cannot be violated outside its own operations. This approach has a great advantage: the users of a component can assume its invariant without having to prove it. But, B users must deal with important architecture restrictions that ensure the soundness of reasonings involving invariants. Moreover, understanding how these restrictions ensure soundness is not trivial. This paper studies a meta-model of invariant composition, inspired from the Spec# approach. Basically, in this model, invariant violations are monitored using ghost variables. The consistency of assumptions about invariants is controlled by very simple proof obligations. Hence, this model provides a simple framework to understand B composition rules and to study some conservative extensions of B authorizing more architectures and providing more control on components initialization.

1 Introduction

Approaches based on formal specifications or annotations become widespread. They are based on specifications by contract [18] and invariants [13]. When components like modules or objects, are involved, the notion of invariant requires a careful attention, both in specification and validation processes. In particular, several issues must be addressed [21]: which variables may an invariant depend on? when do invariants have to hold? It has be proved that a very lax approach, as initially adopted in JML [14,15], cannot be really implemented in static verifiers. Indeed, when components are involved, verification is also expected to be modular: invariants must be established without examining the entire program. Consequently, specifiers need a methodology to explicitly reason about invariants and their preservation in layered architectures.

In the B method [1], architectural restrictions ensure that when components are combined, their respective invariants are preserved without further proof obligations. Hence, in a well-formed architecture, invariant propagation and verification is a transparent process for developers. The simplicity of invariant composition and the control of proof obligations through composition are the main features in industrial uses of the B method. For example, the Météor project [2,9]

J. Julliand and O. Kouchnarenko (Eds.): B 2006, LNCS 4355, pp. 4–18, 2006.

involves about 1000 components, while keeping manageable the proof process. New projects as Roissy Val [3] and Line 1 of Paris involve even more components. For railway applications, constructors have developed methodological guides to build architectures which are adapted to the domain and fulfill architectural restrictions. The B method has also been used in the domain of smart card applications [17,24,5]. But, as detailed in section 5, reconciling B restrictions and natural architectures of applications is harder in this domain.

Alternatively, other approaches, like Spec# [16,6], are based on an explicit treatment of invariants validity. In the Spec# approach, invariants are properties that hold except when they are declared to be broken. The main difficulty of this approach is the specification overheads: developers must describe which invariants are expected to hold. Thus, the specification process becomes much more complex and error-prone. In this paper, we propose to relax architectural restrictions of B using Spec# ideas. But, we avoid Spec# specification overheads, by characterizing some patterns of architectures.

Section 2 introduces the principles of invariant composition in the B method and restrictions associated to this approach. Section 3 presents a meta-model of invariant composition, inspired from the Spec# approach. Section 4 shows how the invariant composition of the B method can be explained from our meta-model. Finally, section 5 proposes a new invariant composition principle, and illustrates the proposed approach through a case study.

2 A Brief Presentation of the B Method

At first, we recall some basic notions about the B method. The core language of B specifications is based on three formalisms: data are specified using a set theory, properties are first order predicates and the behavioral part is specified by *Generalized Substitutions*. Generalized Substitutions can be defined by the Weakest Precondition (WP) semantics, introduced by E.W. Dijkstra [10], and denoted here by $[S]R$. Here are two WP definition examples:

$[\ \text{PRE}\ P\ \ \text{THEN}\ S\ \ \text{END}]\, R \Leftrightarrow P\ \wedge\ [S]\, R$	pre-conditioned substitution
$[S_1\ ;\ S_2]\, R \Leftrightarrow [S_1]\, [S_2]\, R$	sequential substitution

Generalized substitutions can equivalently be characterized by two predicates, $\mathsf{trm}(S)$ and $\mathsf{prd}(S)$, that respectively express the required condition for substitution S to terminate, and the relation between before and after states (denoted respectively by v and v'). Weakest precondition calculus includes termination: we have $[S]R \Rightarrow \mathsf{trm}(S)$, for any R.

Definition 1 (trm and prd predicates)

$$\mathsf{trm}(S) \Leftrightarrow [S]true \qquad\qquad \mathsf{prd}_v(S) \Leftrightarrow \neg[S]\neg(v' = v)$$

2.1 Abstract Machine

As proposed by C. Morgan [19], B components correspond to the standard notion of state machines which define an initial state and a set of operations, acting

on internal state variables. Moreover, an invariant is attached to an abstract machine: this is a property which must hold in observables states, i.e. states before and after operations calls. Roughly, an abstract machine has the following shape[1]:

```
MACHINE M
VARIABLES v
INVARIANT I
INITIALIZATION U
OPERATIONS
   o ← nom_op(i) ≙
      PRE P THEN S END ;
   ...
END
```

M is the component name, v a list of variable names, I a property on variables v, and U a generalized substitution. In the operation definition, i (resp. o) denotes the list of input (resp. output) parameters, P is the precondition on v and i, and S is a generalized substitution which describes how v and o are updated.

Proof obligations attached to machine M consist in showing that I is an inductive property of component M:

Definition 2 (Invariant proof obligations)

$$\begin{array}{lll} (1) & [U]I & \textit{initialization} \\ (2) & I \ \wedge \ P \Rightarrow [S]I & \textit{operations} \end{array}$$

2.2 Invariants Composition

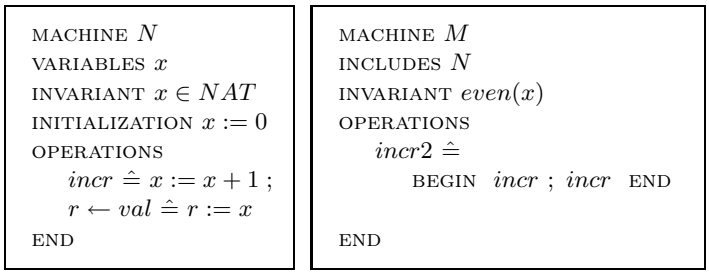
Abstract machines can be combined, through the two primitives INCLUDES and SEES to build new specifications. We do not consider here the clause USES, which is not really used and supported by tools.

The first feature underlying invariant composition in the B method is invariant preservation by encapsulated substitutions. A substitution S is an encapsulated substitution relative to a given component M if and only if variables of M are not directly assigned in S, but only through calls to M operations. Thus, any encapsulated substitution relative to M preserves, by construction, invariant of M (see [23]). The second feature underlying invariant composition is a set of restrictions when components are combined together:

- M INCLUDES N means that operations of M can be defined using any N operations and M invariant can constrain N variables.
- M SEES N means that operations of M can only call read-only N operations and M invariant can not constrain N variables.

Moreover there is no cycle in INCLUDES and SEES dependencies and INCLUDES dependency relation is a tree (each machine can be included only once). These restrictions prevent from combining operations that constrain shared variables in inconsistent ways. Let us consider the following example:

[1] Some others rubrics are permitted such as constants, ... Because they have no particular effect on the composition process, we do not take them into account here.



(1)	$init$; $incr2$; $r \leftarrow val$; $incr2$;	admitted by B
(2)	$init$; $incr2$; $incr$; $incr2$;	rejected by B
(3)	$init$; $incr2$; $incr$; $incr$; $incr2$;	rejected by B

Sequence 1 is authorized because it combines a read-only operation of N with operations of M. Sequences 2 and 3 are rejected because they combine modifying operations of N and M. The reject of sequence 2 prevents the second call of operation $incr2$ to occur in a state where invariant of M is broken. But, sequence 3 is rejected although each operation-call happens in a state where all invariants visible from this operation are valid.

In practical experiments, these restrictions make the design of architectures difficult [4,12]. Actually, components can share variables only in a very limited way (at most one writer-several readers). Section 5 describes a smart card case study [5] illustrating problems raised by a real application.

3 A Meta-model for B Components Inspired from Spec#

The Spec# approach [6,16] proposes a flexible methodology for modular verification of objects invariants, which is based on a dynamic notion of ownership. An ownership relation describes which objects can constrain others objects, i.e. which object has an invariant depending on the value of another object. It is imposed that this relation is dynamically a forest. This allows to generate proof obligations ensuring that an object is not modified while it is constrained by the invariant of an other object. Dynamic ownership of an object can be transfered during execution, introducing some flexibility with respect to B restrictions. We directly present the Spec# approach in the framework of B components and generalized substitutions style.

Hence, this section proposes a new module language for B inspired from the Spec# approach. In section 4, this module language is considered as a kind of meta-model in which we interpret each composition mechanisms of B. This allows to check very simply the soundness of B proofs obligations with respect to components composition. And in section 5, we study how this meta-model can be used to relax B restrictions on composition. In the present state of our work, refinement of B is not considered, and let for future works.

3.1 Static Ownership and Admissible Invariants

In our module language, we first assume a relation *owns* between components: we write $(M, N) \in owns$ to express that "M *owns* N". This relation *owns* is called *static ownership* and is related to admissible invariants, i.e. the invariants which are not automatically rejected by the static analyzer. By definition 3 below, *admissible invariants* of M can only have free variables bound to components transitively owned by M. In the following, $M.\mathsf{Inv}$ denotes the invariant stated in component M, and $M.\mathsf{Var}$ denotes the set of variables declared in M, and $free(P)$ denotes the set of unbound variables appearing in formula P, and $owns^*$ is the reflexive and transitive closure of relation *owns*.

Definition 3 (Admissible invariant through the *owns* relation)

$$free(M.\mathsf{Inv}) \subseteq \bigcup_{N \in\ owns^*[M]}\ N.\mathsf{Var}$$

Here, this notion of admissible invariant is the unique assumption about *owns*: correct instances of the meta-model must define *owns* such that their notion of admissible invariant matches exactly definition 3. In section 4, we show that if we consider that *owns* corresponds exactly to INCLUDES clauses between B components, then B is a quite simple instance of the meta-model. In section 5, in order to relax B restrictions related to INCLUDES clauses, we propose to consider *owns* only as a particular sub-relation of INCLUDES.

Actually, static ownership is related to validity of invariants: the invariant of a component M can not be *safely assumed*, when the state of a component transitively owned by M has been modified outside of M scope. Moreover, we will see that static ownership gives a hierarchical structure to validity of invariants: if the invariant of component M can be safely assumed, then all the invariants of components transitively owned by M can also be safely assumed.

3.2 Dynamic Ownership and Ghost Status Variable

Our module language controls the consistency of constraints on a component M by ensuring that in each state of the execution, M has at most one unique *dynamic owner*. Hence, validity of invariants is now precised: the invariant of a component X constraining an other component M can be safely assumed, only when X transitively *owns* a dynamic owner of M.

In order to express this control, each component M contains a ghost status variable, called M.st, and belonging to $\{\texttt{invalid}, \texttt{valid}, \texttt{committed}\}$. By definition, a component X is a *dynamic owner* of M if and only if $(X, M) \in owns$ and $X.\text{st} \neq \texttt{invalid}$. And, for each component M, M.st is intended to satisfy:

- if $M.\text{st} = \texttt{invalid}$ then $M.\mathsf{Inv}$ may be false. In particular, any modification on M variables is authorized. Moreover, M has no dynamic owner.
- if $M.\text{st} = \texttt{valid}$ then $M.\mathsf{Inv}$ is established and M has no dynamic owner.
- if $M.\text{st} = \texttt{committed}$ then $M.\mathsf{Inv}$ is established and M has a single dynamic owner.

More formally, for each component M, variable $M.\mathrm{st}$ has to verify the following meta-invariants:

$\mathcal{MI}_1$	$M.\mathrm{st} \neq \texttt{invalid} \Rightarrow M.\mathsf{Inv}$
$\mathcal{MI}_2$	$M.\mathrm{st} \neq \texttt{invalid} \wedge (M,N) \in owns \Rightarrow N.\mathrm{st} = \texttt{committed}$
$\mathcal{MI}_3$	$M.\mathrm{st} = \texttt{committed} \wedge\ (A,M) \in owns \wedge (B,M) \in owns$ $\wedge\ A.\mathrm{st} \neq \texttt{invalid} \wedge B.\mathrm{st} \neq \texttt{invalid} \Rightarrow A = B$

The first meta-invariant states that a component invariant can be safely assumed if its status is different from `invalid`. Meta-invariant $\mathcal{MI}_2$ imposes that when a component invariant is not `invalid` then components transitively owned by this component have to be declared as `committed`. Finally $\mathcal{MI}_3$ ensures that a component has at most one unique dynamic owner.

3.3 Preconditioned Assignment Substitution

In our module language, assignment substitution is preconditioned, but it can occur outside of the component where the assigned variable is bound (there is *a priori* no variable encapsulation). We have:

subst	**trm**	**prd**
$N.\mathsf{Var} := e$	$N.st = \texttt{invalid}$	$N.st' = N.st \wedge\ N.\mathsf{Var}' = e$

Meta-invariants $\mathcal{MI}_2$ and $\mathcal{MI}_3$ are obviously preserved by this substitution, because status variables remain unchanged. We want to prove that the assignment substitution preserves the meta-invariant $\mathcal{MI}_1$ for any component M, i.e.:

$$M.\mathrm{st} \neq \texttt{invalid} \ \Rightarrow\ (N.\mathrm{st} = \texttt{invalid} \ \Rightarrow\ [N.\mathsf{Var} := e]M.\mathsf{Inv})$$

There are three cases:

1. if $N = M$ then $\mathcal{MI}_1$ holds because hypotheses are contradictory.
2. if $(M,N) \in owns^+$ then the two hypotheses $N.\mathrm{st} = \texttt{invalid}$ and $M.\mathrm{st} \neq \texttt{invalid}$ are also contradictory due to $\mathcal{MI}_2$: $N.\mathrm{st}$ must be equal to `committed`.
3. if $(M,N) \notin owns^*$ then, due to definition 3, $N.\mathsf{Var} \cap free(M.\mathsf{Inv}) = \emptyset$. In this case $M.\mathsf{Inv}$ is obviously preserved by the assignment substitution $N.\mathsf{Var} := e$.

3.4 `pack`(M) and `unpack`(M) Substitutions

Substitutions are extended with two new commands `pack`(M) and `unpack`(M). The former requires the establishment of M invariant and the latter allows violation of M invariant. Status variables can only be modified via these commands. They can be invoked in M or outside of M. They are formally defined by:

subst	trm	prd
$\texttt{pack}(M)$	$\forall N.((M,N)\in owns \Rightarrow N.\mathrm{st}=\texttt{valid})$ $\wedge\ M.\mathrm{st} = \texttt{invalid}$ $\wedge\ M.\mathsf{Inv}$	$\forall N.((M,N)\in owns \Rightarrow N.\mathrm{st}'=\texttt{committed})$ $\forall N.(M\neq N \wedge (M,N)\notin owns \Rightarrow N.\mathrm{st}'=N.\mathrm{st})$ $\wedge\ M.\mathrm{st}' = \texttt{valid}$ $\wedge\ M.\mathsf{Var}' = M.\mathsf{Var}$
$\texttt{unpack}(M)$	$M.\mathrm{st} = \texttt{valid}$	$\forall N.((M,N)\in owns \Rightarrow N.\mathrm{st}'=\texttt{valid})$ $\forall N.(M\neq N \wedge (M,N)\notin owns \Rightarrow N.\mathrm{st}'=N.\mathrm{st})$ $\wedge\ M.\mathrm{st}' = \texttt{invalid}$ $\wedge\ M.\mathsf{Var}' = M.\mathsf{Var}$

Control of dynamic ownership appears in these commands. The precondition of $\texttt{pack}(M)$ imposes that the components statically owned by M have no dynamic owners, then $\texttt{pack}(M)$ makes M the dynamic owner of all the components that it statically owns. Of course, $\texttt{unpack}(M)$ has the reverse effect. Here, let us note that the precondition of $\texttt{unpack}(M)$ imposes that M has itself no dynamic owner. In other words, if we want to unpack a component N in order to modify it through a preconditioned assignment, we are first obliged to unpack its dynamic owner (and so on recursively). It is easy to prove that meta-invariants $\mathcal{MI}_1$, $\mathcal{MI}_2$ and $\mathcal{MI}_3$ are preserved by these `pack` and `unpack`.

Finally, each substitution S built from preconditioned assignment, `pack` and `unpack` constructors and other standard B substitutions satisfies proposition 1.

Proposition 1 (Meta-invariants preservation)

$$\mathcal{MI}_1 \wedge\ \mathcal{MI}_2 \wedge\ \mathcal{MI}_3 \wedge\ \mathsf{trm}(S) \Rightarrow\ [S](\mathcal{MI}_1 \wedge\ \mathcal{MI}_2 \wedge\ \mathcal{MI}_3)$$

3.5 Revisiting Example of Section 2.2

substitution S_e	condition	status modification
$incr2$;	$M.\mathrm{st} = \texttt{valid}$	
$\texttt{unpack}(M)$;	$M.\mathrm{st} = \texttt{valid}$	$M.\mathrm{st} := \texttt{invalid} \| \| N.\mathrm{st} := \texttt{valid}$
$incr$;	$N.\mathrm{st} = \texttt{valid}$	
$incr$;	$N.\mathrm{st} = \texttt{valid}$	
$\texttt{pack}(M)$;	$N.\mathrm{st} = \texttt{valid} \wedge M.\mathsf{Inv} \wedge M.\mathrm{st} = \texttt{invalid}$	$M.\mathrm{st} := \texttt{valid} \| \| N.\mathrm{st} := \texttt{committed}$
$incr2$;	$M.\mathrm{st} = \texttt{valid}$	

In our meta-model, sequence 3 of example section 2.2 can be extended by `pack` and `unpack` substitutions such that if S_e denotes the resulting substitution, then we can prove $\mathsf{trm}(S_e)$. In the table above, we have represented the proof obligation generated for $\mathsf{trm}(S_e)$ by associating each basic step of the sequence to its precondition and its modification of the environment.

In order to express the B semantics of components M and N of the example into our meta-model, we impose that $(M, N) \in owns$ and that substitutions of modifying operations defined in a component X are implicitly bracketed by

`unpack`(X) and `pack`(X). Hence, in calls to modifying operations of X, variable X.st is required to be `valid` before the call, and is ensured to be `valid` after the call. Moreover, we impose that this sequence starts in a state such that M.st = `valid` and N.st = `committed`. This interpretation of the B language is justified in section 4.

4 Encoding B Operations in This Meta-model

This section proposes to check the consistency of invariants in B architectures, by embedding B components into our meta-model, and verifying that proofs obligations of B allow to discharge the proof obligations from the meta-model.

First, we consider that $(M, N) \in owns$ if and only if M contains a clause "INCLUDES N". Indeed, clause SEES can not be included in the ownership relation defined section 3, because it does not authorize seen variables to be constrained by the invariant part. So, we consider a relation *sees*, such that $(M, N) \in sees$ if and only if M contains a clause SEES N. Moreover, if $(M, N) \in sees$, there is no meta-invariant like $\mathcal{MI}_2$ ensuring that validity of M invariant implies validity of N invariant. Indeed, modifying operations of N can be called while M is valid: this leads to an intermediary state where N is unpacked, but M not. Hence, below, the status of seen components is managed in preconditions of operations.

At last, we need to label B operations using status variables, according to implicit conditions of B with respect to component invariants. We distinguish three cases: the initialization process, operations belonging to component interfaces and local operations.

4.1 Initialization Process

In the B method, global initialization is an internal process which sequentializes local initializations in an order compatible with component dependencies. This step is analog to the Ada `elaborate` phase and is tool-dependent: depending on the architecture, several orders could be possible, resulting in different initial values. This global process can be described in the following way: let $<$ be a partial order defined by $N < M$ if and only if $(M, N) \in owns \cup sees$. From $<$ a total order is built giving in that an initialization procedure specified in the following way:

PRE $\forall N \ . (N \in COMP \Rightarrow N.\text{st} = \texttt{invalid})$
THEN
  U_1 ; `pack`(C_1) ;
  ...
  U_n ; `pack`(C_n) ;
END

with C_i denoting the component labeled by i in the total order and U_i the initialization substitution of component C_i. $COMP$ represents the set of components concerned by the global initialization process. We proved that this initialization procedure terminates, and establishes meta-invariants $\mathcal{MI}_1$, $\mathcal{MI}_2$ and $\mathcal{MI}_3$ (thanks to proposition 1).

4.2 Interface Operations

In the same way, B operations of a component M can be labeled by status information. We consider two forms of operations: modifying operations and read-only operations. Let PRE P THEN S END be the definition of an operation belonging to interface of M. This operation can be labeled in the following way:

modifying case	PRE $P \wedge\ M.\mathrm{st} = \texttt{valid}$ $\wedge\ \forall\ N\ .((M,N) \in (owns \cup sees)^+ \Rightarrow N.\mathrm{st} \neq \texttt{invalid})$ THEN $\texttt{unpack}(M)\ ;\ S\ ;\ \texttt{pack}(M)$ END
read-only case	PRE $P \wedge\ M.\mathrm{st} \neq \texttt{invalid}$ $\wedge\ \forall\ N\ .((M,N) \in (owns \cup sees)^+ \Rightarrow N.\mathrm{st} \neq \texttt{invalid})$ THEN S END

In the modifying case, direct assignments of M variables can occur because M is unpacked. For read-only operations, status precondition is weakest because a read-only operation does not contain assignment. Hence, read-only operations can be called, even if M is committed. Formula $\forall\ N\ .((M,N) \in (owns \cup sees)^+ \Rightarrow N.\mathrm{st} \neq \texttt{invalid})$ preconditions, guarantees that invariants of transitively seen components are also valid, as said at the beginning of this section.

As before, proof obligations of termination contain proof obligations of B invariants, corresponding to termination of `pack` calls. Finally, if S does not contain explicit `pack` and `unpack` substitutions and respects B restrictions (M variables can be directly assigned, included variables are only assignable through operation calls and seen variables can not be modified in any way) then all status conditions are established by construction. In particular, any operation body S of component M fulfills the following property, for any component N:

Proposition 2 (Status preservation through B operations)

$$\mathcal{MI}_1 \wedge\ \mathcal{MI}_2 \wedge\ \mathcal{MI}_3 \wedge\ \mathsf{prd}(S) \Rightarrow\ N.\mathrm{st}' = N.\mathrm{st}$$

4.3 Local Operations

In B, local operations can be introduced at the level of implementation [8]. They authorize assignment of component variables as well as direct assignments of included variables[2]. Local operations can be seen as private operations allowing to factorize code. They do not have to preserve local invariant. Proof obligations given in [8] consists in proving the preservation of all invariants of included components. So, let M be a component which (transitively) includes components $N_1, \ldots, N_n$. A local operation in M, defined by the substitution PRE P THEN S END, can be labeled[3] in the following way:

[2] At the level of implementation, inclusion takes the form of an IMPORTS clause. Here we do not distinguish these two mechanisms, as it is done in B for the clause SEES.

[3] To simplify, we do not take into account seen components.

PRE $P \wedge M.\mathrm{st} = \texttt{invalid} \wedge N_1.\mathrm{st} = \texttt{valid} \wedge \ldots \wedge N_n.\mathrm{st} = \texttt{valid}$
THEN $\texttt{unpack}(N_1)$; ... ; $\texttt{unpack}(N_n)$;
S ;
$\texttt{pack}(N_1)$; ... ; $\texttt{pack}(N_n)$;
END

Remark that several other forms of local operations would be possible. For instance, we should define friendly local operations that must be called in a state where the local invariant holds. Such operations would not require to repeat the invariant (or some part of it) in the precondition of a local operation, as it is often necessary. Similarly, developers could choose which variables are assigned and so which included components needs to be unpacked. This would avoid to reproved all imported invariants: this may be interesting even if such proofs are obvious. To introduce such flexibility we now define an extension of B language allowing to directly manipulate conditions on status variables and substitutions `pack` and `unpack`.

5 A More Flexible Composition Principle

Modifying the previous interpretation of B in our meta-model, this section proposes to extend B with a more flexible invariant composition principle, allowing to specify more sharing between components and more natural architectures. Specifications can now directly reference component status variables but only in precondition or assertion parts. Moreover, specifications can perform `unpack` and `pack` substitutions. By this way, developers can precisely state when invariants must hold. Now, invariant preservation proof obligation is no more an external process, but directly integrated into the language. Let PRE P THEN S END be an operation definition. Now, its proof obligation is:

$$(\mathcal{MI}_1 \wedge \mathcal{MI}_2 \wedge \mathcal{MI}_3 \wedge P) \Rightarrow \mathsf{trm}(S)$$

Such a proof obligation guarantees the consistent use of status variables. Moreover, if S contains some `pack` substitutions then the termination proof obligation contains proofs obligations relative to the validity of the expected invariants.

Finally, to obtain a powerful invariant composition principle, we propose a more flexible initialization process and a smaller notion of ownership than the B INCLUDES relation.

5.1 Initialization and Reinitialization Process

As stated section 4.1, in the B method initialization is a global process. Because more sharing is now admitted, we make explicit the initialization process in using operations which establish invariants (on the contrary to interface operations which preserve invariant). In this way, the developer can specify an initialization order in a precise way, avoiding uncontrollable non-determinism of B initialization process. At last, initializations can be invoked in any place, in order to

reinitialize variables. Let U be the initialization substitution of component M. Several form of initialization operations are possible:

case 1	PRE M.st = `invalid` THEN U ; `pack`(M) END
case 2	PRE M.st = `invalid` $\wedge$ N.st $\neq$ `invalid` THEN U ; `pack`(M) END
case 3	PRE M.st = `invalid` $\wedge$ N.st = `invalid` THEN $N.Init$; U ; `pack`(M) END

Case 1 corresponds to initialization of a stand-alone machine. Case 2 corresponds to an initialization depending on another initialization, like when M sees N in B. Case 3 corresponds to an initialization performing another initialization: here, $N.Init$ denotes an initialization operation of component N. Case 3 implicitly happens in B when M includes N: in particular, the elaboration of M invariant involves initial values of N.

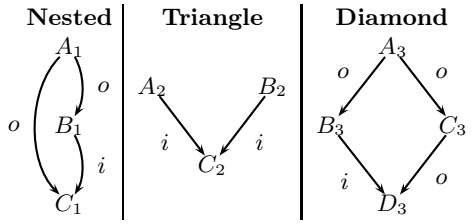
5.2 A Smaller *owns* Relation

Modular invariant composition is based on two orthogonal aspects: on one hand, it is necessary to control components that constrain shared variables and, on the other hand, it is also necessary to control assignment of shared variables. In the Spec# approach, the ownership relation allows to control in which way variables are constrained and the preconditioned assignment substitution allows to control when sets of variables can be updated. In B, these two aspects are syntactically grouped together through the two clauses clauses SEES and INCLUDES:

	read operation	modifying operation
variables can be constrained	INCLUDES	INCLUDES
variables cannot be constrained	SEES	-

The case uncovered by B corresponds to components that call any operations of a given component M but do not constrain its variables. In this case, there is no problem as soon as the shared component is not committed (all other components with an invariant depending on these variables are invalid). So, we narrow relation *owns* to only keep the subrelation corresponding to components whose variables are effectively strengthened by new invariants. Formally, we now define the *owns* relation as the smallest subrelation of INCLUDES$^+$ (the transitive closure of the relation induced by INCLUDES clauses) respecting the notion of admissible invariant (def. 3). Such a relation exists and is unique. All results of section 3 apply here.

With this smaller *owns* relation, we now have more meaningful architectures than before. Let us compare three approaches. Approach 1 is strict B, based on static restrictions about architectures. Approach 2 is the approach of section 4, in which the *owns* relation is assimilated to the clause INCLUDES. At last, approach 3 corresponds to the smaller definition of *owns* given above. We consider



the three architectures given below. In the two first approaches, both kind of arrows represent an INCLUDES clause. In approach 3, *o*-arrows represent an *owns* pair, and *i*-arrows represent a INCLUDES pair that is not a *owns* pair.

In approach 1, all these architectures are rejected. In approach 2, architecture `Nested1` is meaningless, because A_1 can be never packed. Indeed, B_1 must be packed before A_1, but when B_1 is packed, C_1 is committed, and A_1 can not be packed. On the contrary, in approach 3, A_1 can be packed because C_1 is not committed by B_1. Moreover, interface operations of B_1 can call interface operations of C_1. In approach 2, architecture `Triangle` can be used with restrictions: one of the two components A_2 or B_2 must be invalid because C_2 can be committed only once. In approach 3, C_2 is never committed, thus A_2 and B_2 can be packed and can freely call interface operations of C_2. Finally, architecture `Diamond` is also meaningless in approach 2 (A_3 can not be packed). On the contrary, in approach 3, interface operations of B_3 can call interface operations of D_3 when C_3 is invalid. In A_3, we may thus need to unpack C_3 before to call operation of B_3 and then repack C_3.

5.3 A Case Study

We present here an example extracted from the Java Card byte-code interpreter case study developed in the BOM project [FME'03]. A Java card virtual machine has a four components architecture:

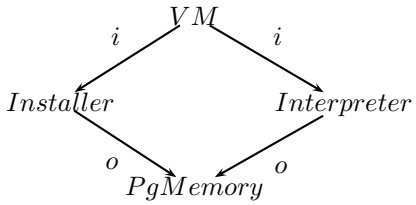
- $PgMemory$ contains the byte-code of the program to run. It performs an abstraction of the physical memory.
- $Installer$ performs the loading of the byte-code into the memory of the card.
- $Interpreter$ provides operations corresponding to each instruction of the virtual machine.
- VM is the main component, it provides an operation to install a program and run it step-by-step, using the operations of $Interpreter$.

In Java Card, the format of byte-code is required to satisfy well-formness properties that guarantee safety and security properties of the execution [20]: for instance, a valid instruction starts by an op-code directly followed by its well-typed parameters, a method entry point always refers an op-code, etc. These properties need to be expressed in the different components of our architecture:

- $PgMemory$: the byte-code stored in memory is well-formed (invariant I_{mem}).
- $Installer$: the byte-code already loaded is well-formed (invariant I_{inst}).

– *Interpreter*: the program counter always points to a valid instruction (invariant I_{interp}).

This architecture can not be directly implemented in B because invariants I_{inst} and I_{interp} constrain the state of $PgMemory$ (see the discussion in the conclusion). On the contrary, in the proposed approach, the diamond architecture on the right is adapted.



In VM, $Installer$ and $Interpreter$ do not need to be valid in the same time: when one is valid, it dynamically owns $PgMemory$ and the other is invalid. So the VM main operation performs the ownership transfer of $PgMemory$ in the following way:

$PgMemory.Init$; $Installer.Init$; $Installer.load$;
unpack($Installer$) ;
$Interpreter.Init$; $Interpreter.exec$

6 Conclusion

The approach proposed here allows to express invariants which are only valid on some portions of programs. Such invariants can also be expressed in B by predicate of the form $co = i \Rightarrow I_i$ where co is a variable simulating an "ordinal counter" and I_i the expected invariant when $co = i$. Such forms of invariants are extensively used in B-event development. Nevertheless, this solution is not modular because variable co can not be updated by several components. For instance, a B solution for the case study of section 5.3 consists in building an architecture where $Installer$ includes $PgMemory$, $Interpreter$ sees $PgMemory$ and VM includes both $Installer$ and $Interpreter$. The invariant I_{interp} is stated at the level of component VM, in the form $co = interp \Rightarrow I_{interp}$. This needs to add the precondition $co = interp$ to each operation of $Interpreter$ (see [12] for a general solution). Variable co is assigned by VM, but, each $Interpreter$ operation-call requires precondition $co = interp$ to be established. As stated in [4], this solution forces to specify some expected invariants in a top component, rather than in components where operations concerned by these invariants are defined. Moreover, invariants of the form $co = i \Rightarrow I_i$ may be confusing because such invariants should never be violated, but strongly depend on the control. The Spec# approach systematizes this solution, by introducing explicitly variables and statements relative to invariants validity. Hence, the notion of invariant validity is more explicit, while preserving soundness and modularity of invariant proofs.

The adaptation of the Spec# approach to B proposed in this paper leads to very simple proof obligations about status variables, most of them being obvious. Nevertheless, on the contrary to the B method in which invariant composition

is a transparent process, the Spec# approach is very permissive making specification and proof an hard task. Hence, it is necessary to propose some patterns having good properties, like the different form of B operations described section 4. Moreover, it seems also possible to define a static analysis allowing to approximate whether status variables are used in a consistent way.

Other approaches have been studied in order to overcome B restrictions and in particular the single writer constraint. In [4], a rely-guarantee approach has been proposed in order to support architectures where two components A and B both need to constrain variables of C and to write into these variables. Basically, in this approach, the user must express in C what A and B are authorized to do on variables of C. Hence, both A and B know an abstraction of the other behavior on C, and can verify that this behavior is compatible with their own invariants. This approach is thus compatible with refinement (in particular, refinements of A and B refine their respective abstraction with respect to C). Our approach seems suitable for the multiple writers paradigm only when all ownership transfers between successive writers are performed via a reinitialization operation. But, in the other cases, when interface operations of writers are interleaved without reinitialization, then our approach is not modular with respect to the previous rely-guarantee approach: each ownership transfer requires a proof that the invariant of the new owner hold. On the contrary, the rely-guarantee approach of [4] does not authorize some combinations which are permitted by our approach. Indeed, our approach does not impose that invariants of all writers hold concurrently for all possible interleavings. Hence, it would be interesting to study the extension of our approach with rely-guarantee. Some proposals in this direction have already been studied for Spec# by Naumann and Barnett [7,22].

Finally, the main characteristics of the B method are its notion of component refinement and the monotonicity property allowing substitution of operation specifications by their implementations. Technically, refinement in B is based on invariants [11]: the relations between abstract and concrete data are expressed in a "gluing" invariant of the refining component. If we want to mix B refinement with our approach, we have the following problem: when we use some `pack`(M) outside of component M, we are obliged to prove that the invariant of M holds, but also that invariants of all refinements of M hold in order to ensure the substitutability principle. Let us remark that when `pack`(M) is used inside an operation of M, this problem does not occur: the proofs that gluing invariants hold are done in refining operations. Moreover, `unpack`(M) may occur outside of M without problem. Thus, in practice, it seems that refinement is compatible with our approach when it is restricted such that `pack`(M) calls occur only in component M. The case study of section 5 is an example where our approach accepts an architecture not admitted B, which is still compatible with refinement.

References

1. J.R. Abrial. *The B-Book.* Cambridge University Press, 1996.
2. P. Behm and all. Météor: A Successful Application of B in a Large Project. In *FM'99*, vol 1708 of *LNCS*, pages 348–387. Springer-Verlag, 1999.

3. F. Badeau and A. Amelot. Using B in a High Level Programming Language in an Industrial Project : Roissy VAL. In *ZB 2005*, vol 3455 of *LNCS*. Springer-Verlag, 2005.
4. M. Büchi and R. Back. Compositional Symmetric Sharing in B. In *FM'99*, vol 1708 of *LNCS*. Springer-Verlag, 1999.
5. D. Bert, S. Boulmé, M-L. Potet, A. Requet, and L. Voisin. Adaptable Translator of B Specifications to Embedded C programs. In *FME 2003: Formal Methods*, vol 2805 of *LNCS*. Springer, 2003.
6. M. Barnett, R. DeLine, M. Fähndrich, K.R.M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.
7. M. Barnett and D.A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *MPC*, vol 3125 of *LNCS*. Springer, 2004.
8. ClearSy. *Le Langage B. Manuel de reference, version 1.8.5*. ClearSy, 2002.
9. D.Dollé, D. Essamé, and J. Falampin. B à Siemens Transportation Systems- Une expérience industrielle. In *Développement rigoureux de logiciel avec la méthode B*, vol 22. Technique et Science Informatiques, 2003.
10. E.W. Dijkstra. *A discipline of Programming*. Prentice-Hall, 1976.
11. D. Gries and J. Prins. A New Notion of Encapsulation. In *Proc. of Symp. on Languages Issues in Programming Environments, SIGLPAN*, 1985.
12. H. Habrias. *Spécification formelle avec B*. Hermès Science, 2001.
13. C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.
14. G. Leavens, A. Baker, and C. Ruby. JML: A notation for detailed design. In *Behavioral Specifications of Businesses and Systems*. Kluwer Academic Publishers, 1999.
15. G. Leavens, A.L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical report, TR 98-06i, Iowa State University, 2000.
16. K.R.M. Leino and P. Müller. Object invariants in dynamic contexts. In *ECOOP*, vol 3086 of *LNCS*, pages 491–516. Springer-Verlag, 2004.
17. J-L. Lanet and A. Requet. Formal Proof of Smart Card Applets Correctness. In *CARDIS'98*, vol 1820 of *LNCS*. Springer-Verlag, 1998.
18. B. Meyer. *Object-Oriented Construction*. Prentice-Hall, 1988.
19. C. Morgan and P.H.B. Gardiner. Data Refinement by Calculation. *Acta Informatica*, 27(6):481–503, 1990.
20. Sun Microsystems. Java Card^{TM} 2.2 Off-Card Verifier. Tech. report, Sun microsystems, 901 San Antonio Road, Palo Alto, CA 94303 USA, June 2002.
21. P. Müller, A. Poetzsch-Heffer, and G.T. Leavens. Modular Invariants for Layered Object Structures. *Science of Computer Programming*, 2006.
22. D.A. Naumann and M. Barnett. Towards imperative modules: Reasoning about invariants and sharing of mutable state. In *LICS 2004*. IEEE Computer Society, 2004.
23. M-L. Potet. *Spécifications et développements formels: Etude des aspects compositionnels dans la méthode B*. Habilitation à diriger des recherches, Institut National Polytechnique de Grenoble, 5 décembre 2002.
24. D. Sabatier and P. Lartigue. The Use of the B method for the Design and the Validation of the Transaction Mechanism for smart Card Applications. *Formal Methods in System Design*, 17(3):245–272, 2000.