

Relaxing restrictions on invariant composition in the B method by ownership control *a la* SPEC#

Sylvain Boulmé, Marie-Laure Potet

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

Abstract. This paper deals with modular verification of component invariants in the method B. On one side, B imposes severe architecture restrictions that ensure soundness of component compositions without additional proof obligations. On the other side, in the context of the verification of object oriented programs, SPEC# proposes a more expressive approach, but at the price of more complex specifications, and more numerous proof obligations. In this paper, we investigate an intermediate solution combining advantages of both approaches.

1 Introduction

A project (or a development) in the method B is constituted of several components, layered in abstract machine, refinements and implementation. A major feature of this component language is that each component can be developed and proved independently. Architecture restrictions of B ensure that composition of components is free: it does not need additional proofs. This is one of the keys that make the method B scalable for large industrial applications.

In particular, at the level of invariants, this property is ensured because the invariant of a component can not be violated outside of its own operations. Hence, the users of a component M can assume the invariant of M without to prove it. This invariant is established by the proofs obligations induced by the definition of M , independently of other components. More precisely, concerning components that are used by M , the proof obligations in M ensure the preservation of M invariant, only for the particular uses made by M of these components. Thus, if the invariant of M constrains variables of a component N , then a third component could *a priori* call an operation of N , that modifies the variable of N such that the invariant of M is violated. Actually, such a case *can not happen* in B because of severe architecture restrictions.

In summary, if B users can compose components without reproving their invariants, they 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 investigates a very simple (meta)model of invariant composition, inspired from SPEC# approach. This model is based on a dynamic notion of *ownership* that characterizes which components can constrain other components through an invariant. Technically, invariant violations are monitored using ghost variables. The consistency of assumptions about invariants with respect to the ownership dependencies is controlled by proof obligations involving these ghost variables. However, in our simplified model with respect to SPEC#, these proof obligations are very simple and can be discharged by a dedicated static analyzer.

Hence, this model provides a simple framework to understand B composition rules. And, it also inspires us some conservative extensions of B which authorize more architectures and provide a better control on the initialization process of components. In such extensions, current developments in B would be still valid without new proof obligations.

This paper is organized as follows. Section 2 presents the B method. In particular, it gives some architecture restrictions of B and illustrates them on an example. Section 3 presents our simple adaptation of SPEC# for B. It explains how this approach extends the expressive power of B. Section 4 illustrate these ideas on a little case study. Section 5 presents our plans to make a conservative extension of B allowing the control of ownership *a la* SPEC#. The main idea is to consider the ghost variables introduced by SPEC# methodology as a *type information* and to replace proof obligations involving these variables by *type inference*. Hence, current developments of B could be embedded without having to insert annotations manually, and more important, without new proof obligations. At last, section 6 presents other perspectives: refinement and extensions with rely-guarantee.

2 A short presentation of B

This section introduces some background about B specifications and proof obligation process. It first presents the core specification language of B, called *the language of generalized substitution*. Then, it presents *abstract machines* : what are their associated proof obligations, how they can be composed, what are the restrictions on these compositions.

2.1 Generalized substitution

B specifications are based on three formalisms: data are specified using a set theory, properties are First Order Predicate Calculus formulas and the behavioral part is specified by *Generalized Substitutions*.

The generalized substitution language offers a set of constructions in an imperative form. This choice proves to be well-adapted for proof activity, because assignment is directly interpreted as a substitution command. In the following we only introduce constructions which will be useful in this paper.

Notation	Name
$x := e$	simple substitution
skip	substitution with no effect
$P \mid S$	pre-conditioned substitution
$P \Longrightarrow S$	guarded substitution
$@z \cdot S$	unbounded choice substitution
$S_1 ; S_2$	sequential substitution
$S_1 \parallel S_2$	choice substitution

Generalized substitutions can be seen as predicate transformers. As introduced by E.W. Dijkstra [Dij76], they can be defined by the Weakest Precondition (*WP*) semantics, denoted here by $[S]R$. The following array gives the *WP* definition for the considered basic substitutions.

Definition 1 (Basic Axioms).

<i>Axiom</i>	<i>Condition</i>
$[\text{skip}] R \Leftrightarrow R$	
$[P \mid S] R \Leftrightarrow P \wedge [S] R$	
$[P \Longrightarrow S] R \Leftrightarrow P \Rightarrow [S] R$	
$[@z \cdot S] R \Leftrightarrow \forall z \cdot [S] R$	$z \notin \text{free}(R)$
$[S_1 ; S_2] R \Leftrightarrow [S_1] [S_2] R$	
$[S_1 \parallel S_2] R \Leftrightarrow [S_1] R \wedge [S_2] R$	

$[x := e]R$ reduces to the substitution of free occurrences of x by e in the predicate R . The notation $\text{free}(e)$ denotes the free variables of the expression or predicate e . Predicate $\text{trm}(S)$ characterizes value for which substitution S terminates.

Definition 2 (trm and pred predicates).

$$\begin{aligned} \text{trm}(S) &\Leftrightarrow [S]\text{true} \\ \text{prd}(S) &\Leftrightarrow \neg[S]\neg(x' = x) \end{aligned}$$

Weakest precondition calculus includes termination. We have $[S]R \Rightarrow \text{trm}(S)$, for any R .

2.2 Abstract Machine

As proposed by C. Morgan [MG90], **B** components correspond to the classical notion of state machines which define an initial state and a set of operations, acting on internal state variables. Moreover, a notion of invariant is attached to an abstract machine. The invariant property is relative to the set of operations: it is related to observables states, i.e. states before and after operations calls. Roughly, an abstract machine has the following shape¹:

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

MACHINE M
VARIABLES v
INVARIANT I
INITIALIZATION U
OPERATIONS
$o \leftarrow \text{nom_op}(i) \hat{=} P \mid S ;$
...
END

M is the component name, v a list of variable names and I a logical property on variables v . Initialization is a generalized substitution U with an empty set of input variables. In the operation definition, i denotes the list of input parameters, o the list of output parameters. The formula P is the precondition and only depends on variables v and i . S is a generalized substitution which describes how v and o are updated.

Proof obligations consist in showing that I is an inductive property of component M . The invariant must be established by the initialization and preserved by each operation of the component, when the precondition holds. Proof obligations are obtained by the following formulas:

Definition 3 (Invariant proof obligations).

- (1) $[U]I$ *initialization*
- (2) $I \wedge P \Rightarrow [S]I$ *operations*

2.3 Abstract machines and invariants composition

Abstract machines can be combined, through the two primitives INCLUDES and SEES to build new specifications. Combinations are subjected to strict rules, in order to combine invariants without any further proof obligations. As pointed out in the introduction part, compositional proofs is an important property to develop large projects in keeping the proof process manageable. The first feature underlying invariant composition in the B method is the invariant preservation when an abstract machine is extended.

- Invariant proof obligations are relative to operation and are preserved by operations call due to the following theorem:
Let $r \leftarrow \text{op}(p) \hat{=} P \mid S$ be an operation definition and let $v \leftarrow \text{op}(e)$ be a call with a copy semantics. Then:

$$\begin{aligned} & \forall r, p \quad (I \wedge P \Rightarrow [S]I) \\ & \Rightarrow \\ & (I \wedge [p := e]P \Rightarrow [@ p, r . p := e ; S ; v := r]I) \end{aligned}$$

- Components are used in an encapsulated way through generalized substitutions which preserve, by construction, invariants. For instance we have:

$$\frac{I \wedge \text{trm}(U_1) \Rightarrow [U_1]I \quad I \wedge \text{trm}(U_2) \Rightarrow [U_2]I}{I \wedge \text{trm}(U_1 ; U_2) \Rightarrow [U_1 ; U_2]I}$$

The second feature is a set of restrictions when components are combined together:

- M INCLUDES N means that operations of M can be defined by using any N operations and M invariant can constrained N variables
- M SEES N means that operations of M can be defined by using only read-only N operations and M invariant cannot constrain N variables
- there is no cycle in INCLUDES and SEES dependencies
- in a structured component INCLUDES dependency relation is a tree (each machine can be included only once)

Fundamental property underlying these clauses is it is not possible to combine operations which constrained shared variables in different ways. Let's consider the following example:

Example 1 (Composing operations coming from several components).

<pre> MACHINE M VARIABLES x INVARIANT $x \in NAT$ INITIALIZATION $x := 0$ OPERATIONS $incr \hat{=} x := x + 1 ;$ $r \leftarrow val \hat{=} r := x$ END </pre>	<pre> MACHINE M_2 INCLUDES M INVARIANT $even(x)$ OPERATIONS $incr2 \hat{=}$ BEGIN $incr ; incr$ END END </pre>
---	---

(1) $init ; incr2 ; r \leftarrow val ; incr2 ;$	OK for B
(2) $init ; incr2 ; incr ; incr2 ;$	KO for B
(3) $init ; incr2 ; incr ; incr ; incr2 ;$	KO for B

Sequence 1 is authorized because it combines only a read-only operation of M (N INCLUDES M and N SEES M_2). Sequence 2 is refused because it combines modifying operations of M and M_2 (N INCLUDES M and N INCLUDES M_2). In this sequence the second call of operation $incr2$ takes place in a state where invariant of M_2 is broken. Sequence 3 is also refused because it combines modifying operations of M and M_2 , although each operation calls takes place in states where their invariants are valid.

As pointed out by the last example B restrictions can be too restrictive. In practical experiences, architectures can be hard to build and do not necessarily correspond to natural architectures ([BB99,Hab01]) because they strongly depend on properties which must be stated (admissible invariants in clauses INCLUDES or SEES). Finally, components can share variables only in a very limited way (at most one writer-several readers). A case study illustrating these problems is described in section 4.

3 Adapting SPEC# approach for modules *a la* B

The SPEC# approach [BDF⁺04,LM04] proposes a flexible methodology for modular verification of objects invariants, based on a dynamic notion of ownership. An ownership relation describes which objects can constrain others (sub)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. In this way a (sub)object can only be constrained by another one and invariants can be preserved. Ownership relation can be transferred during execution, allowing in that flexibility. The main characteristics of this approach are the following:

- authorizes invariants violation outside the scope of objects definition, but in a controlled way.
- safe use of objects, with respect to their invariant status, is *monitored* by proofs.

Technically, each component has a ghost field, indicating *dynamically* the status of its invariant. These ghost variable allows in the same time to abstract the status of invariants (violated or established) and to put a lock on a component, like in concurrent programming. When a component is locked (or *committed*): this component is informed that another component X constrains it, thus it forbids to modify its variable because invariant of X may be violated.

We directly present our adaptation of the SPEC# approach in the framework of B components and generalized substitutions style.

3.1 Dependencies between components and admissible invariants

In this paper, we impose that components declare their dependency on others components using `depends` clause. Syntactic restrictions ensure that the relation `depends`⁺ between components is irreflexive. Hence, `depends` relation makes components structured as a directed acyclic graph. Moreover, if two components A and B depend on a third one C , then C is *shared* by A and B : any modification performed by A on C is seen from B .

In practice, we may imagine that a component A is allowed to have a private copy of a component B . This feature does not interfere with our form of dependency: it can be achieved as a renaming mechanism.

When A depends transitively on B , A can call operations of B , read or write variables of B and constrain B in its assertions or in its invariant. More formally, we define below the notion of admissible invariants.

Definition 4 (Admissible invariant). *An invariant of a component M is syntactically admissible if and only if all the free variable appearing in this invariant belongs to a component in $\text{depends}^*(\{M\})$, the image of M through the reflexive and transitive closure of `depends`.*

In the following we denote by $M.\text{Inv}$ the invariant stated in component M and by $M.\text{Var}$ variables declared in M . The previous definition says that $M.\text{Inv}$

is admissible iff

$$free(M.Inv) \subseteq \bigcup_{N \in \text{depends}^*(\{M\})} N.Var$$

3.2 Ownership and ghost status variable

Actually, there are two notions of ownership: a static one and a dynamic one. Static ownership is only required to be acyclic, whereas dynamic ownership is required to be a forest (e.g. without sharing).

The static ownership relation, denoted by `owns`, is a subrelation of `depends+` that must satisfy that for all admissible invariants $M.Inv$, then

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

In a first approximation, the reader can imagine that `owns` equals to `depends`. However, we will see in subsection 3.7, that in order to get a maximal expressibility, `owns` has to be more carefully defined.

Each component M is implicitly extended by a ghost variable $M.st$, belonging to `{invalid, valid, committed}`. A component N is a *dynamic owner* of a component M if and only if $(N, M) \in \text{owns}$ and $N.st \neq \text{invalid}$. The semantics of this ghost variable is the following:

- if $M.st = \text{invalid}$, then its invariant may be violated. Moreover, M has no dynamic owner. Hence, any modification on M variables is authorized.
- if $M.st = \text{valid}$, then its invariant is established, and it has no dynamic owner.
- if $M.st = \text{committed}$, then its invariant is established, and it has a single dynamic owner.

For each component M , variables `st` have to verify the following meta-invariants (the conjunction of these three meta-invariants is noted \mathcal{MI}):

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

The first invariant states that, to determine if a component invariant holds, it is sufficient to check if its status is different from `invalid`. \mathcal{MI}_2 imposes that when a component invariant is not `invalid` then components which are directly or indirectly owned by it have to be declared as `committed`. From \mathcal{MI}_1 and \mathcal{MI}_2 we have the following expected property:

Property 1 (Hierarchical invariants). For each component M , when invariant of M holds then invariants of all components transitively owned by M also hold.

Finally \mathcal{MI}_3 ensures that a component can dynamically belong to at most one other component. As seen below \mathcal{MI}_3 will be necessary to establish \mathcal{MI}_2 preservation.

3.3 Extending the language of generalized substitution

Assignment substitution At first assignment substitution is now preconditioned. We have:

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

Assignment is the most sensitive substitution because it modifies variables and can invalidate invariants. We have to prove that the assignment substitution preserve meta-invariant \mathcal{MI}_1 for any component M , i.e.:

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

There are three cases to take into consideration: $N = M$, $(M, N) \in \text{owns}^+$ and $(M, N) \notin \text{owns}^+ \wedge N \neq M$.

1. \mathcal{MI}_1 holds because hypotheses are contradictory.
2. if $(M, N) \in \text{owns}^+$ then the two hypotheses $N.\text{st} = \text{invalid}$ and $M.\text{st} \neq \text{invalid}$ are also contradictory, due to meta-invariant \mathcal{MI}_2 ($N.\text{st}$ must be equal to `committed`).
3. if $(M, N) \notin \text{owns}^+$ and $M \neq N$ then $N.\text{Var} \cap \text{free}(M.\text{Inv}) = \emptyset$ (this is a consequence of the property $\text{free}(M.\text{Inv}) \subseteq \bigcup_{N \in \text{owns}^+(\{M\})} N.\text{Var}$) Then $M.\text{Inv}$ is obviously preserved.

Meta-invariants \mathcal{MI}_2 and \mathcal{MI}_3 are obviously preserved because status remain unchanged.

pack(M) and unpack(M) substitutions. Substitutions are extended with two new commands `pack(M)` and `unpack(M)`. The first one requires the establishment of an invariant and the second one allows violation of an invariant. Only these commands directly modify `st` variables. They are formally defined by:

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

We proved that meta-invariants \mathcal{MI}_1 , \mathcal{MI}_2 and \mathcal{MI}_3 are preserved by these two substitutions. In particular:

- $\text{pack}(M)$ preserves \mathcal{MI}_3 , because \mathcal{MI}_2 ensures that in the pre-state, every N statically owned by M has no dynamic owner ($N.\text{st}$ being `valid`).
- $\text{unpack}(M)$ preserves \mathcal{MI}_2 , because \mathcal{MI}_3 ensures that in the pre-state, every N statically owned by M has only M as dynamic owner.

Finally for each substitution S built from preconditioned assignment, `pack` and `unpack` new constructors and other classical B substitutions (except assignment substitution) we have:

Proposition 1 (Meta-invariants preservation).

$$\mathcal{MI} \wedge \text{trm}(S) \Rightarrow [S]\mathcal{MI}$$

3.4 Proof obligations for definitions of operations

The meaning of components is given through the proof obligations attach to the definition of their operations:

- Initially $M.\text{st} = \text{invalid}$, for all components M (other variables of M are initialized non-deterministically by a value in their type). Hence, the architecture must have a main operation, where all components are required to be `invalid`.
- All operations `PRE P THEN S END` must satisfy $\mathcal{MI} \wedge P \Rightarrow \text{trm}(S)$ where
 - `st` variables can occur free in P
 - S can perform `unpack` and `pack`

In other words, preservation of components invariants must be expressed explicitly through `unpack` and `pack` command.

Here, initializations are particular operations which can be directly invoked by developers: an initialization in a component M is simply an operation that have a precondition $M.st = \text{invalid}$ and a postcondition $M.st = \text{valid}$. Hence, this operation performs a $\text{pack}(M)$.

This gives more flexibility than \mathbf{B} , a needed flexibility when more sharing is admitted. Indeed, now, initializations can have parameter and preconditions. Moreover, the developer can specify an precise initialization order, avoiding uncontrollable non-determinism of \mathbf{B} initialization process. At last, initializations can be invoked anywhere, in order to reinitialize variables and repack a component.

Actually, we have shown than there is an encoding of \mathbf{B} architectures without refinements in this system such that proof obligations of \mathbf{B} can be deduced from the proof obligations of this approach. For instance, we call *interface operations of a component M* , operations with a body of the form $\text{unpack}(M); S; \text{pack}(M)$ such that there is no unpack or pack command inside S . Interface operations correspond exactly to operations of \mathbf{B} .

3.5 Revisiting example 1

Encoding the example 1 in our system (\mathbf{B} operations being encoded as interface operations), we can now label the sequence 3 with pack and unpack substitutions such that termination (trm) of this sequence is provable in our system. We here suppose this sequence starts in a state such that $M_2.st = \text{valid}$ and $M.st = \text{committed}$ (such a state holds after the encoding of \mathbf{B} initialization process that we have not detailed here).

substitution	status modification	condition
$\text{incr2} ;$		$M_2.st = \text{valid}$
$\text{unpack}(M_2) ;$	$M_2.st := \text{invalid} M.st := \text{valid}$	$M_2.st = \text{valid}$
$\text{incr} ;$		$M.st = \text{valid}$
$\text{incr} ;$		$M.st = \text{valid}$
$\text{pack}(M_2) ;$	$M_2.st := \text{valid} M.st := \text{committed}$	$M.st = \text{valid}$
		$\bigwedge M_2.\text{Inv}$
		$\bigwedge M_2.st = \text{invalid}$
$\text{incr2} ;$		$M_2.st = \text{valid}$

3.6 Restrictions on architectures in SPEC# methodology

At this point, it appears that owns relation is an extension of INCLUDES clause of \mathbf{B} . But, what kind of architectures can we have that are not present in \mathbf{B} ? First, let us try to understand restrictions imposed in practice by SPEC# methodology. We illustrate these restrictions on architectures of figure 1, where each node represent a component, and each arrow correspond to a owns pair.

Let us first examine the constraints on the Nested1 architecture:

- Using the notion of admissible invariant of previous sections, B_1 can constrain C_1 in its invariant, and A_1 can constrain both B_1 and C_1 .

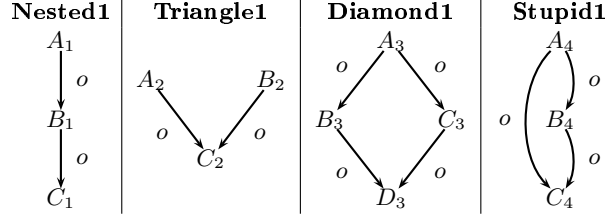


Fig. 1. Example of architectures with owns only

- The successive packing (e.g. the initialization) of all components can be performed: C_1 is first packed, then B_1 is packed, and at last A_1 is packed.
- Interface operations of B_1 can call interface operations of C_1 .
- Interface operations of A_1 can call interface operations of B_1 . But, they can not directly call interface operations of C_1 . Because, when B_1 is valid, then C_1 is committed. Hence, in order to call a interface operation of C_1 in A_1 , B_1 must be unpacked first.

We recover here exactly the constraints of **B** with respect to **INCLUDES** clause.

In the Triangle1 architecture (rejected in **B**), A_2 and B_2 can not be valid in the same time. Indeed, by the meta-invariant, when A_2 is valid then C_2 is committed and B_2 must be invalid.

As a consequence, in the Diamond1 architecture (also rejected in **B**), A_3 can never be valid: this architecture is thus not very meaningful.

At last, in the Stupid1 architecture, A_4 can never be valid. In this architecture, the set of admissible invariants is not changed with respect to the Nested1 architecture. Hence, Nested1 is much better.

These examples shows that in the SPEC# methodology, the **owns** relation must be carefully chosen. If it is too large, then it forbids to pack some components. If it is too small, then it forbids some invariants.

3.7 A fine owns relation

In order to support more architectures than **B**, it seems thus interesting to define a fine **owns** relation: if M depends on N without constraining it, then M may not need to own N . In particular, such a relation between M and N would authorize M to call interface operations of N when N is not committed.

Hence now, we define the **owns** relation as a sub-relation of **depends*** having the *smallest* number of elements which satisfies “for all admissible invariants $M.Inv$, $free(M.Inv) \subseteq \bigcup_{N \in owns^*({M})} N.Var$ ”. Such a relation exists and is unique. This can be shown by induction over **depends***.

Moreover, the proof of the meta-invariant page is still correct. And previous remarks about architecture of figure 1 are also still valid (except for the Stupid1 architecture which does not fit the definition of **owns**).

However, by having this finer owns relation, we can now have more meaningful architectures than before (and than B). In particular, given $(M, N) \in \text{depends}^*$ such that $\text{free}(M.\text{Inv}) \cap N.\text{Var} = \emptyset$, then $(M, N) \notin \text{owns}$ (but we may have that $(M, N) \in \text{owns}^+$, if we are in a nested architecture).

We now study the architecture of figure 2, where arrows labeled with o represent a owns pair, and arrows labeled with d represent a depends pair that is not a owns pair.

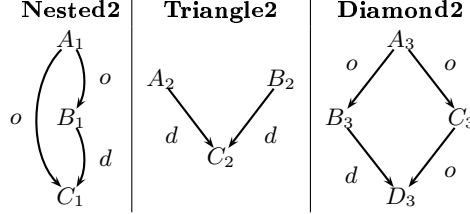


Fig. 2. Meaningful architectures with the finer owns relation

Using proof obligations of SPEC# methodology, let us first examine the difference between Nested1 of figure 1 and Nested2 of figure 2.

- In Nested2, B_1 does not constrain C_1 in its invariant (whereas in Nested1, it can do so).
- In Nested2, interface operations of A_1 can call both interface operations of C_1 and B_1 . Indeed, interface operations of B_1 can only call interface operations of C_1 when C_1 is not committed. But, this is the case in interface operation of A_1 , because A_1 has been unpacked.

Actually, B authorizes only a restricted form of Nested2, when B_1 sees C_1 (B_1 can thus only call read-only operations of C_1).

In architecture Triangle2, C_2 is never committed, thus A_2 and B_2 can be packed, and can freely call interface operations of C_2 .

In architecture Diamond2, A_3 can be packed. Interface operations of B_3 can call interface operations of D_3 only 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 .

4 Case study

In this section, we propose to study an invented but simplified example which reveals architecture problems that appear to us in real case studies, like the javacard bytecode interpreter developed in the BOM project [FME'03]. Hence, let us imagine a simple virtual machine having naturally a four components architecture:

- *Memory* is the core component to read or write data in memory. Indeed, it performs an abstraction of the physical memory. In particular, this component contains the bytecode of the program to run.
- *Installer* performs the loading of the bytecode transmitted by a terminal into the memory of the card.
- *Interpreter* provides operations corresponding to each instructions 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 javacard, the format of bytecode is required to satisfy well-formedness properties that guarantee safety and security properties of the execution. These properties are statically verified (usually off-card).

- method entry point refers an op-code
- byte codes are valid: each op-code is followed by its well-typed parameters
- a bytecode instruction cannot be shared between several methods
- etc.

These properties need to be expressed in the different components of our architecture:

- the bytecode in *Memory* is well-formed.
- During the loading of the bytecode in *Installer*, the bytecode already loaded is well-formed.
- In *Interpreter*, the program counter always points to a valid instruction.
- etc.

Let us now sum up the constraints on the architecture:

- *Installer* and *Interpreter* need both to write into *Memory*.
- *Installer* and *Interpreter* need both to express an invariant relating their variables to those of *Memory*.
- *VM* need to call write operations of *Installer* and *Interpreter*. In *VM* calls first *Installer* and then *Interpreter*.

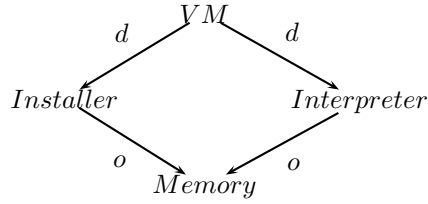
Actually, no B architecture can deal satisfactorily with such an architecture:

- *Installer* and *Interpreter* can not both includes *Memory*.
- if *Interpreter* includes *Installer* (or conversely), it can not call write operations of *Memory*.

The only solution is to make a single components for *Installer*, *Interpreter* and *VM*. But this is not modular.

However, this architecture can be expressed in our system: *Interpreter* and *Installer* statically owns *Memory*, and *VM* depends on *Installer* and *Interpreter*. In *VM*, *Installer* and *Interpreter* are neither valid in the same time:

when one is valid, it dynamically owns *Memory* and the other is invalid. We are in the following diamond architecture:



The delicate point in this architecture is thus the ownership transfer of *Memory* from *Installer* to *Interpreter* in component *VM* shown below. This example also illustrates the use of initialization operations that we have described section 3.4.

transfer of *Memory* ownership

$sp := \text{Installer.load};$ `unpack(Installer) ; Interpreter.start(sp)`

Operation *start* of *Interpreter* requires that *Memory* is valid (and thus initialized) and initializes *Interpreter* variables and packs it. For instance, variable *ip* (the instruction pointer which refers to the next address corresponding to an instruction to run) is initialized from a parameter *sp* (the address of the first instruction to run). But invariant *I* expresses that *ip* must point to a valid opcode. Hence, to ensure this property at the end of initialization, *start* requires a precondition *P* on memory variables: *sp* is a valid opcode. This precondition *P* of *Interpreter.start* must thus be a postcondition of the *Installer.load* execution.

```

MACHINE Interpreter
INCLUDES Memory
VARIABLES ip, ...
INVARIANT I
OPERATIONS start(sp) =
  PRE Memory.st = valid  $\wedge$  P THEN
    ip := sp ;
    pack(Interpreter)
  END
...
  
```

where $\mathcal{MI} \wedge \text{Memory.st} = \text{valid} \wedge P \Rightarrow [ip := sp]I$

This example has a noticeable property: in *VM*, during the ownership transfer of *Memory* from *Installer* to *Interpreter*, the whole *Interpreter* invariant has not to be proved directly, but only an abstraction of this invariant: *P*. In other words, there is no `pack` instruction here. This `pack` instruction is performed inside *Interpreter.start*, and thus, the proof of *Interpreter* invariant is done

at the definition of this operation. Hence, each component of the architecture can be independently refined without breaking the soundness of this ownership transfer.

5 Static analysis and proof obligations

An important drawback of the SPEC# approach, is that the user has to provide the preconditions on component status. This is tedious and error prone. Indeed, the user can give preconditions which are unsound with respect to the meta-invariant. And, the unsoundness may be revealed only at the end, in the proof of termination (*trm*) of the main operation under the hypothesis that all components are (initially) invalid. Moreover, SPEC# methodology generates a lot of proof obligations, which are mostly trivial.

Hence, we propose to check the consistency of invariant violations using a static analysis instead of proof obligations. Moreover, our static analysis must infer preconditions over component status in operations and reject preventively operations which could be unsound with respect to the meta-invariant. We can benefit here from the fact that, in **B**, on the contrary to SPEC#, there is no alias on component names.

Basically, with this static analysis, we hope to be able to propose a conservative extension to **B**, such that current developments in **B** could be embedded without having to insert annotations manually, and more important, without new proof obligations. Of course, it would be needed to insert `unpack/pack` and status typing assertions, but these insertions could be performed automatically.

5.1 Requirements on the static analysis

Actually, such a static analysis needs to restrict a bit the expressive power of our language, but with few practical consequences. Indeed, it requires that if a substitution is a branch (a choice, an if-then-else, a while-loop), then all branches perform the same observational transformation on the status variables. For instance, if a branch is `skip`, others branches can only perform well-bracketed `unpack/pack` or `pack/unpack`. Actually, this seems rather healthy. Furthermore, guards, which are not allowed to express properties about `st` anymore, are just skipped by the static analysis.

In the SPEC# approach, given an operation of body $P|S$, we have to prove:

$$MI \wedge P \Rightarrow \text{trm}(S)$$

Most often, this proposition P is written $Q \wedge P'$ where Q does not involve `st` variables and P' only involves `st` variables. Here, we would like to discharge the user of the boring task of writing P' . This practical consideration leads to the main idea of our static analysis:

1. the weakest-precondition calculus is changed in a new calculus, written $[S]^\dagger R$ which does not involve `st` variable anymore.

2. we try to abstract S in a substitution $\mathcal{P}!A$ where \mathcal{P} is only a conjunction of atomic clauses about `st` variables, and A is only a multiple affectation about `st` variables.

If our abstraction algorithm of S fails, nothing can be ensured: S may have unsound preconditions on `st` variable, but it is not certain. If our abstraction algorithm of S succeeds, then:

$$(\mathcal{P} \wedge \mathcal{MI} \wedge [S]^\dagger[A]R) \Rightarrow [S]R$$

Thus, we have in particular that:

$$(Q \Rightarrow [S]^\dagger True) \Rightarrow (\mathcal{MI} \wedge Q \wedge \mathcal{P} \Rightarrow \text{trm}(S))$$

Hence, the proof obligation $Q \Rightarrow [S]^\dagger True$ is sufficient in order to ensure preservation of meta-invariant \mathcal{MI} . Moreover, the syntax of \mathcal{P} ensures in particular that satisfiability of formula $\mathcal{MI} \wedge \mathcal{P}$ is decidable.

Below, we illustrate the expected behavior of our typing rules on a few examples.

Example 2 (Basic typing). Assuming 3 components A, B, C such that A owns B and B owns C , then :

- substitution $\text{unpack}(A); B.x := e$ must be rejected. Indeed, status precondition of $B.x := e$ is $B.\text{st} \in \{\text{invalid}\}$ whereas status affectation of $\text{unpack}(A)$ is $A.\text{st} := \text{invalid} || B.\text{st} := \text{valid}$.
- substitution $\text{unpack}(A); \text{unpack}(B)$ must be abstracted in $A.\text{st} \in \{\text{valid}\} | (A.\text{st} := \text{invalid} || B.\text{st} := \text{invalid} || C.\text{st} := \text{valid})$.

Example 3 (Typing approximations). Assuming a components A then :

- substitution $\text{skip} || (\text{unpack}(A); \text{pack}(A))$ must be abstracted in $A.\text{st} \in \{\text{valid}\} | \text{skip}$.
- substitution $\text{skip} || \text{unpack}(A)$ is rejected because the two branches perform incompatible observational effects on status.
- substitution $False \implies \text{pack}(A)$ is abstracted in $A.\text{st} \in \{\text{invalid}\} | A.\text{st} := \text{valid}$.

Example 4 (Precondition unsoundness). Assuming 3 components A, B, C such that A owns C and B owns C , then substitution $\text{unpack}(A); \text{unpack}(B)$ is abstracted in $(A.\text{st} \in \{\text{valid}\} \wedge B.\text{st} \in \{\text{valid}\}) | (A.\text{st} := \text{invalid} || B.\text{st} := \text{invalid} || C.\text{st} := \text{valid})$. However, the inferred status precondition is unsound with respect to the meta-invariant. Hence this substitution must be finally rejected.

5.2 A first sketch of the static analysis

In order to convince the reader that such an analysis is feasible, we propose here a very simple one. Below, we now consider `st` as typing information associated to each components. And each substitution S is required to have a type $\mathcal{P}!A$, where

$$\frac{}{\overline{True \sqcap \mathcal{P} \rightsquigarrow \mathcal{P}}} \qquad \frac{\mathcal{P}_1 \sqcap \mathcal{P}_2 \rightsquigarrow \mathcal{P}_3 \quad M \notin FV(\mathcal{P}_2)}{\overline{(M.st \in St \wedge \mathcal{P}_1) \sqcap \mathcal{P}_2 \rightsquigarrow M.st \in St \wedge \mathcal{P}_3}}$$

$$\frac{St_1 \cap St_2 \rightsquigarrow St_3 \quad \mathcal{P}_1 \sqcap \mathcal{P}_2 \rightsquigarrow \mathcal{P}_3}{\overline{(M.st \in St_1 \wedge \mathcal{P}_1) \sqcap (M.st \in St_2 \wedge \mathcal{P}_2) \rightsquigarrow (M.st \in St_3) \wedge \mathcal{P}_3}}$$

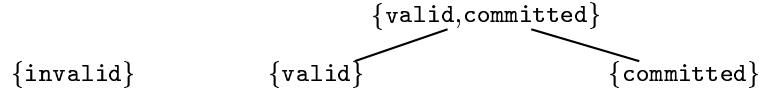
Fig. 3. Definition of $\mathcal{P}_1 \sqcap \mathcal{P}_2 \rightsquigarrow \mathcal{P}_3$ (conjunction of 2 status preconditions)

$$\frac{}{\overline{[\text{skip}]\mathcal{P} \rightsquigarrow \mathcal{P}}} \qquad \frac{[\mathcal{A}]\mathcal{P}_1 \rightsquigarrow \mathcal{P}_2 \quad M \notin FV(\mathcal{P}_1)}{\overline{[M.st := v|\mathcal{A}]\mathcal{P}_1 \rightsquigarrow \mathcal{P}_2}}$$

$$\frac{v \in St \quad [\mathcal{A}]\mathcal{P}_1 \rightsquigarrow \mathcal{P}_2}{\overline{[M.st := v|\mathcal{A}](M.st \in St \wedge \mathcal{P}_1) \rightsquigarrow \mathcal{P}_2}}$$

Fig. 4. Definition of $[\mathcal{A}]\mathcal{P}_1 \rightsquigarrow \mathcal{P}_2$ (substitution of an affectation in a status precondition)

- \mathcal{P} is a *status precondition*. It expresses a precondition on the status components. Syntactically, it is either $True$ or $M.st \in St \wedge \mathcal{P}$, where component name M does not appear in \mathcal{P} and St belongs to the following partial order:



We write $St_1 \cap St_2 \rightsquigarrow St$ when the intersection of St_1 and St_2 exists and is St (given any St_1 and any St_2 , their intersection does not always exist).

- \mathcal{A} is a *status affectation*. It expresses an affectation on the status of components. Syntactically, \mathcal{A} is either skip or $M.st := v|\mathcal{A}$ where component name M does not appear in \mathcal{A} , and v is *invalid*, *valid* or *committed*.

We call *status types* these substitution types. Of course, the syntax chosen here suggests that for status types, $\mathcal{P}!\mathcal{A}$ can be interpreted as a substitution $\mathcal{P}|\mathcal{A}$, working on variable st of components. We use this interpretation to give the semantics of our type system.

We need also to change the language of substitutions : substitutions are not allowed to use st variable in assertions, except in dedicated preconditions written $\mathcal{P}!S$. In our new weakest-precondition calculus (written $[S]^\dagger R$), pack and unpack do not involve variable st anymore, but only the invariant of components : pack sets the invariant as a precondition, whereas unpack sets it as a postcondition.

Hence, the semantics of our type system is that if a substitution S admits a type $\mathcal{P}!\mathcal{A}$, we write this $S \rightsquigarrow \mathcal{P}!\mathcal{A}$, then for all R ,

$$(\mathcal{P} \wedge [S]^\dagger[\mathcal{A}]R) \wedge MI \Rightarrow [S]R$$

Another important property of our type-system, is that if $S \rightsquigarrow \mathcal{P}!\mathcal{A}$, then substitution $\mathcal{P}|\mathcal{A}$ preserves the meta-invariant. In other words, we have:

$$MI \wedge \mathcal{P} \Rightarrow [\mathcal{A}]MI$$

$$\frac{}{\text{skip}; \mathcal{A} \rightsquigarrow \mathcal{A}} \qquad \frac{\mathcal{A}_1; \mathcal{A}_2 \rightsquigarrow \mathcal{A}_3 \quad M \notin FV(\mathcal{A}_2)}{(M.\text{st} := v || \mathcal{A}_1); \mathcal{A}_2 \rightsquigarrow M.\text{st} := v || \mathcal{A}_3}$$

$$\frac{\mathcal{A}_1; \mathcal{A}_2 \rightsquigarrow \mathcal{A}_3}{(M.\text{st} := v_1 || \mathcal{A}_1); (M.\text{st} := v_2 || \mathcal{A}_2) \rightsquigarrow M.\text{st} := v_2 || \mathcal{A}_3}$$

Fig. 5. Definition of $\mathcal{A}_1; \mathcal{A}_2 \rightsquigarrow \mathcal{A}_3$ (sequence of 2 status affectations)

- $\text{skip}_{|\mathcal{P}} \stackrel{def}{=} \text{skip}$
- if there exists \mathcal{P}' such that $\mathcal{P} \equiv M.\text{st} \in \{v\} \wedge \mathcal{P}'$, then $(M.\text{st} := v || \mathcal{A})_{|\mathcal{P}} \stackrel{def}{=} \mathcal{A}_{|\mathcal{P}'}$
- otherwise, $(M.\text{st} := v || \mathcal{A})_{|\mathcal{P}} \stackrel{def}{=} M.\text{st} := v; \mathcal{A}_{|\mathcal{P}}$

Fig. 6. Definition of $\mathcal{A}_{|\mathcal{P}}$ (normalization of a status affectation under a status precondition)

The typing rules given in the following consider status preconditions \mathcal{P} and status affectations \mathcal{A} modulo permutations of atomic clauses: operator \wedge and $||$ are implicitly associative and commutative. The typing rules for inferring status types of substitutions are given figure 7. The typing algorithm is very closed to the algorithm for normalizing substitutions. The main difference concerns the choice (rule **CHOICE**), where both branches are required to produce the same observational effects under the current precondition. Moreover, preconditions and guards (rules **PRE** and **GUARD**) are simply skipped. The typing rules use the following definitions:

- the conjunction of 2 status preconditions (noted $\mathcal{P}_1 \sqcap \mathcal{P}_2 \rightsquigarrow \mathcal{P}_3$) defined figure 3. Semantically, $\mathcal{P}_1 \sqcap \mathcal{P}_2 \rightsquigarrow \mathcal{P}_3$ means that $\mathcal{P}_3 \Leftrightarrow \mathcal{P}_1 \wedge \mathcal{P}_2$.
- the substitution of an affectation in a status precondition (noted $[\mathcal{A}]\mathcal{P}_1 \rightsquigarrow \mathcal{P}_2$) defined figure 4. Semantically, $[\mathcal{A}]\mathcal{P}_1 \rightsquigarrow \mathcal{P}_2$ means that $\mathcal{P}_2 \Leftrightarrow [\mathcal{A}]\mathcal{P}_1$.
- the sequence of 2 status affectations (noted $\mathcal{A}_1; \mathcal{A}_2 \rightsquigarrow \mathcal{A}_3$) defined figure 5.
- the normalization of a status affectation under a status precondition (noted $\mathcal{A}_{|\mathcal{P}}$) defined figure 6. Actually, $\mathcal{A}_{|\mathcal{P}}$ is the affectation involving the least number of components such that $\mathcal{P}!\mathcal{A}_{|\mathcal{P}}$ is semantically equivalent to $\mathcal{P}!\mathcal{A}$.

In conclusion, in this static analysis, conditions and modifications on status variables are inferred from substitutions. The user can also express status preconditions like “ $M.\text{st} \in \{\text{valid}, \text{committed}\}!S$ ” in operations. Such a precondition will be interpreted in the proof of termination, as the hypothesis that the invariant of M holds.

Given an operation $Q|S$, with a status type $\mathcal{P}!\mathcal{A}$, it is decidable to check if \mathcal{P} is sound with respect to the meta-invariant. In other words, we can check if $\mathcal{P} \wedge \mathcal{M}\mathcal{I}$ is satisfiable. Indeed, this formula is satisfiable if and only if:

1. for all components M and N with $(M, N) \in \text{owns}^+$ and for all expressions St_1 and St_2 such that formulas “ $M.\text{st} \in St_1$ ” and formulas “ $N.\text{st} \in St_2$ ”

$$\begin{array}{c}
\frac{}{M.x := e \rightsquigarrow M.\text{st} \in \{\text{invalid}\}!\text{skip}} \mathbf{AFFECT} \\
\\
\frac{\mathcal{P} \equiv M.\text{st} \in \{\text{valid}\} \quad \mathcal{A} \equiv M.\text{st} := \text{invalid} \parallel (\parallel_{N \in \text{owns}(\{M\})} N.\text{st} := \text{valid})}{\text{unpack}(M) \rightsquigarrow \mathcal{P}!\mathcal{A}} \mathbf{UNPACK} \\
\\
\frac{\mathcal{P} \equiv M.\text{st} \in \{\text{invalid}\} \wedge (\bigwedge_{N \in \text{owns}(\{M\})} N.\text{st} \in \{\text{valid}\}) \quad \mathcal{A} \equiv M.\text{st} := \text{valid} \parallel (\parallel_{N \in \text{owns}(\{M\})} N.\text{st} := \text{committed})}{\text{pack}(M) \rightsquigarrow \mathcal{P}!\mathcal{A}} \mathbf{PACK} \\
\\
\frac{S_1 \rightsquigarrow \mathcal{P}_1!\mathcal{A}_1 \quad S_2 \rightsquigarrow \mathcal{P}_2!\mathcal{A}_2 \quad [\mathcal{A}_1]\mathcal{P}_2 \rightsquigarrow \mathcal{P}_3 \quad \mathcal{P}_1 \sqcap \mathcal{P}_3 \rightsquigarrow \mathcal{P} \quad \mathcal{A}_1; \mathcal{A}_2 \rightsquigarrow \mathcal{A}}{S_1; S_2 \rightsquigarrow \mathcal{P}!\mathcal{A}} \mathbf{SEQ} \\
\\
\frac{S_1 \rightsquigarrow \mathcal{P}_1!\mathcal{A}_1 \quad S_2 \rightsquigarrow \mathcal{P}_2!\mathcal{A}_2 \quad \mathcal{P}_1 \sqcap \mathcal{P}_2 \rightsquigarrow \mathcal{P} \quad \mathcal{A}_1|_{\mathcal{P}} \equiv \mathcal{A}_2|_{\mathcal{P}}}{S_1 \parallel S_2 \rightsquigarrow \mathcal{P}!\mathcal{A}_1} \mathbf{CHOICE} \\
\\
\frac{S \rightsquigarrow \mathcal{P}_2!\mathcal{A} \quad \mathcal{P}_1 \sqcap \mathcal{P}_2 \rightsquigarrow \mathcal{P}_3}{\mathcal{P}_1!S \rightsquigarrow \mathcal{P}_3!\mathcal{A}} \mathbf{PRE_ST} \\
\\
\frac{S \rightsquigarrow \mathcal{P}!\mathcal{A}}{Q|S \rightsquigarrow \mathcal{P}!\mathcal{A}} \mathbf{PRE} \quad \frac{S \rightsquigarrow \mathcal{P}!\mathcal{A}}{Q \Longrightarrow S \rightsquigarrow \mathcal{P}!\mathcal{A}} \mathbf{GUARD} \quad \frac{S \rightsquigarrow \mathcal{P}!\mathcal{A}}{@z \cdot S \rightsquigarrow \mathcal{P}!\mathcal{A}} \mathbf{ANY}
\end{array}$$

Fig. 7. Definition of $S \rightsquigarrow \mathcal{P}!\mathcal{A}$ (typing rules for substitutions)

- appear in \mathcal{P} , if $St_1 \subseteq \{\text{valid}, \text{committed}\}$ then $St_2 \neq \{\text{valid}\}$ and $St_2 \neq \{\text{invalid}\}$.
2. and, for all components N, A, B with $(A, N) \in \text{owns}^+$ and $(B, N) \in \text{owns}^+$ and $(A, B) \notin \text{owns}^*$ and $(B, A) \notin \text{owns}^*$, and for all expressions St_1 and St_2 such that formulas “ $A.\text{st} \in St_1$ ” and “ $B \in St_2$ ” appear in \mathcal{P} , if $St_1 \subseteq \{\text{valid}, \text{committed}\}$ then $St_2 = \{\text{invalid}\}$.

Our type-system may be improved in the normalization process used by the rule **CHOICE**. Indeed, in this normalization process, it would be more expressive to complete (e.g. saturate) \mathcal{P} with all clauses that can be deduced using $\mathcal{M}\mathcal{Z}$. With our current type-system, the user will sometimes have to annotate the code, using special preconditions $\mathcal{P}!S$, in order to make it typeable.

6 Perspectives

In conclusion, our approach, inspired from $\text{SPEC}\#$ approach, allows to express invariants that are only valid on some portions of programs. Such invariants can also be expressed in **B** by predicate of the form $b = \text{true} \Rightarrow I$ where b is a boolean variable allowing thus to control the validity of I in the program. However, having

such control variables internalized in the module system allows to accept more architectures, while preserving soundness and modularity of invariant proofs.

In future works, we will study two extensions of this approach:

1. an extension with refinement. This is clearly necessary in order to propose a conservative extension to \mathbf{B} .
2. an extension with rely-guarantee. The aim of this extension is to make proofs more modular for multiple writers paradigm.

6.1 Refinement

All properties proved on a machine A are preserved when A is substituted by any of its refinement. This is the *substitutability principle* of refinement. Technically, refinement in \mathbf{B} is based on invariants: the relation between abstract and concrete data are expressed in a “gluing” invariant of the refining component.

If we want to mix these two aspects of \mathbf{B} with our approach, we have the following problem: when we use some $\text{pack}(M)$ outside of component M , we are obliged to prove that 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 $\text{pack}(M)$ is used inside an operation of M , this problem does not occur: the proofs that gluing invariants hold are done in refining operations. Moreover, $\text{unpack}(M)$ 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 $\text{pack}(M)$ occur only in component M . The case study of section 4 is an example where our approach accepts an architecture not admitted \mathbf{B} which is still compatible with refinement.

6.2 Rely-guarantee

Other approaches have been studied in order to overcome \mathbf{B} restrictions and in particular the single writer constraint. For instance, for a given component C , we can not express in \mathbf{B} an architecture where two other components A and B both need to constrain variables of C and to write into these variables. In [BB99], a rely-guarantee approach has been proposed in order to support such an architecture. Basically, in this rely-guarantee approach, the user must express in C what A and B are authorized to do on variables of C . Hence, both A and B knows an abstraction of the other behavior on C , and can verify that this behavior is compatible with their own invariant. 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.

Let us remark however, that the rely guarantee approach of [BB99] does not allow 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 [BN04,NB04].

References

- [BB99] M. Büchi and R. Back. Compositional Symmetric Sharing in B. In J. Davies J.M. Wing, J. Woodcock, editor, *FM'99 - Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.
- [BDF⁺04] Michael Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.
- [BN04] Michael Barnett and David A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *MPC*, pages 54–84, 2004.
- [Dij76] E.W. Dijkstra. *A discipline of Programming*. Prentice-Hall, 1976.
- [Hab01] Henry Habrias. *Spécification formelle avec B*. Hermès Science Publications, 2001.
- [LM04] K.R.M. Leino and P. Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–516. Springer-Verlag, 2004.
- [MG90] C. Morgan and P.H.B. Gardiner. Data Refinement by Calculation. *Acta Informatica*, 27(6):481–503, 1990.
- [NB04] David A. Naumann and Michael Barnett. Towards imperative modules: Reasoning about invariants and sharing of mutable state. In *LICS*, pages 313–323, 2004.