Bringing State Sharing into CSP||B: a B-based Approach *

Samuel Colin[1], Arnaud Lanoix[2], Olga Kouchnarenko[3], and Vincent Poirriez[1]

[1] Univ Lille Nord de France, UVHC, LAMIH, CNRS UMR 8530, F-59313 Valenciennes
scolin@hivernal.org, vincent.poirriez@univ-valenciennes.fr
[2] LINA – UFR de Sciences et Techniques. BP 92208, F-44322 Nantes Cedex
arnaud.lanoix@univ-nantes.fr
[3] LIFC, Univ of Franche-Comté, F-25030 Besançon
okouchnarenko@lifc.univ-fcomte.fr

Abstract. This paper is dedicated to the problem of sharing B machines inside of a CSP||B model. The main obstacle is state sharing: B machines controlled by the various CSP parts are supposed not to refer to, share or modify the same state space. This need for state sharing is motivated by a case study where we want to integrate existing B machines with a shared part into a CSP||B model. To achieve this, we propose an architectural, B-based solution for detecting inconsistencies in CSP||B architectures with state sharing at the B level as well as simplifications for the current consistency checking conditions deducible from our proposal. We explain what consistent models can be established with the proposed criterion. We also hint at possible extensions towards other CSP||B architectural patterns with various types of sub-components sharing and see how our criterion and these extensions would complement existing solutions.

1 Introduction

In this work we address the question of how to safely reuse already-developed B models in which there is a common and shared part when developing a CSP||B model. The problem of sharing is known to be difficult in the framework of the B method whereas it is naturally supported by the CSP formalism.

The present work is motivated by an example which arose during the process of assembling already formally specified and proved components. In the context of the TACOS project, we modelled a situated multi-agent system of a convoy [1] while a complex B model of a location component was also independently designed [2]. Integrating the latter into the former appears to be problematic because the resulting assembly risks breaking the consistency of the whole vehicle component, as state sharing is involved. Machine sharing in the location component is indeed valid at the B level but is not so at the CSP||B level.

Such an architecture goes against the well-known "one controller≡one machine" CSP||B constraint. We are thus interested in lifting this constraint, even only partially,

* Work supported by the ANR-06-SETI-017 project: "TACOS : Trustworthy Assembling of Components: frOm requirements to Specification" (http://tacos.loria.fr).

and in determining under what conditions it can be done. We shall show how using the B modularity constraints and a simplified consistency checking of CSP||B can be used for addressing this problem. Moreover, a developer may *naturally* (or intuitively) design a CSP||B model with multiple controllers for a B machine and also with a single controller for multiple B machines. We give clues for determining whether such architectural patterns would be possible within CSP||B.
Layout of the paper. After a short presentation of the formalisms focused on the concepts relevant for this paper in Sect. 2, we present a simplified version of our problem in Sect. 3. We then relate our problem with similar related work with respect to state sharing in CSP||B and B in Sect. 4. Our main contributions are in Sect. 5, where we propose a methodology based on the B modularity and a simplification of some CSP||B consistency checks requirements for detecting inconsistent CSP||B architectures. We also conjecture at some extensions for addressing the verification of more complex cases. Finally, some conclusions and assessments are drawn in Sect. 6.

2 Concepts and Tools for CSP||B, with a Focus on State Sharing

The B machines specifying components are open modules which interact by the authorized operation invocations. CSP describes processes, i.e. objects or entities which exist independently, but may communicate with each other. When combining CSP and B to develop distributed and concurrent systems, CSP is used to describe execution orders for invoking the B machines operations and communications between the CSP processes. For space reasons, we assume that the readers are familiar with B, CSP and CSP||B. We only focus on the necessary concepts.

2.1 Modularity in B

The whole architecture of a B project (a set of B machines and refinements) must respect certain constraints. For instance, one machine can not end up being included or imported by two different inclusion paths, as this could break the invariant. The modularity constraints of [3] proved to be not strong enough, because intermediate **SEES** links could hide the fact that a machine could be modified through refinement [4]. In [5], modularity constraints for avoiding these problems is given:

Theorem 1. $(uses;can\_alter) \cap ((imports;s^{+}) \cup (sees;s^{*})) = \emptyset$

with *sees* the set of couples (M_1, M_2) where the implementation of M_1 sees the machine M_2, *imports* a similar set where the implementation of M_1 imports M_2, *s* the set where M_1 sees directly M_2, $uses = sees \cup imports$ and $can\_alter = (uses^{*};imports)$. The ; operator corresponds to the B relation composition, * to the B reflexive transitive closure and $^{+}$ to the transitive closure. When taking into account all implicit hypotheses about B modularity [5], the formula can actually be simplified into the following shape:

$$can\_alter \cap sees = \emptyset$$

We introduced this modularity constraint because of the role it plays in our contribution in Sect. 5.

2.2 CSP||B Components

In this section, we sum up the works by Schneider and Treharne on CSP||B. The reader interested in theoretical results is referred to [6,7] and the abundant CSP||B literature referenced therein; for case studies, see for example [8,9].

In CSP||B, the B part is specified as a B machine without any restriction, while the controller is a CSP process, called a CSP controller, defined by the following (subset of the) CSP grammar:

$$\begin{array}{l} \mathsf{P} ::= \mathsf{c}\ ?\ \mathsf{x}\ !\ \mathsf{v}\ \rightarrow \mathsf{P} \mid\ \mathsf{ope}\ !\ \mathsf{v}\ ?\ \mathsf{x} \rightarrow \mathsf{P} \mid\ \mathsf{b}\ \&\ \mathsf{P} \\ \quad\ \mid \mathsf{P} \,\Box\, \mathsf{P} \mid\ \textbf{if}\ \mathsf{b}\ \textbf{then}\ \mathsf{P}\ \textbf{else}\ \mathsf{P} \mid \mathsf{S(p)} \end{array}$$

Machine channels are introduced in CSP controllers to provide the means for controllers to synchronize with the B machine: for each B operation $\mathsf{x} \longleftarrow \mathsf{ope(v)}$, there can be a channel $\mathsf{ope}\ !\ \mathsf{v}\ ?\ \mathsf{x}$ in the controller corresponding to the operation call: the output value v from the CSP description corresponds to the input parameter of the B operation, and the input value x corresponds to the output of the operation. A controlled B machine can only communicate on the machine channels of its controller. An additional requirement for controlled B machines is that they are not allowed to share state, i.e. *see* or *import* the same machines.

When integrating CSP controllers with their B machines into a CSP system, care must be taken with respect to the behaviour of the B machines. Let us assume a CSP controller P and a B machine M associated with P, forming a compound denoted $(\mathsf{P} \| \mathsf{M})$. The verification process to ensure the consistency of $(\mathsf{P} \| \mathsf{M})$ in CSP||B consists in verifying the following conditions: (i) Check the *consistency* of M, with B4Free for instance, (ii) Check the *deadlock-freedom* and *divergence-freedom* of P with FDR2 and (iii) Check the *divergence-freedom* of $(\mathsf{P} \| \mathsf{M})$. With these hypotheses one can deduce the *deadlock-freedom* of $(\mathsf{P} \| \mathsf{M})$ in the stable failures model. A most general result concern the deadlock-freedom of the parallel composition of controlled machines.

Originally, the technique for ensuring the divergence-freedom of a controlled machine involved the stating of a Control Loop Invariant (CLI) and its verification. This technique required some breaking down into as many sub-formulas as there were recursive calls in the controller with the addition of variables tracking in which state the controller was. Fortunately the technique has evolved (see [10,6] for a more in-depth presentation) into a more general and less cumbersome one. Let S, T, S' be states (e.g. predicates expressed in B set theory), e an event corresponding to a B operation, $p(e)(S)(S')$ the property that e is called within its precondition when the before-state is contained in S and the after-state is contained in S' and tr a sequence of operations represented as a list. Then the following $\overrightarrow{every}$ uniform property is introduced:

$$\overrightarrow{every}(p)(S)(T)(tr) = \begin{cases} S \subseteq T & \text{if } tr = nil \\ \exists S'.p(h)(S)(S') \wedge & \\ \qquad \overrightarrow{every}(p)(S')(T)(t) & \text{if } tr = cons(h,t) \end{cases}$$

Let trm and prd be the B operators for calculating the termination and the before-after predicate of a substitution respectively, $op(e)$ be the body of the operation triggered by e, $i(e)$ and $o(e)$ the actual input and output parameters of e, respectively and s a state.

The p property, called an *event predicate*, is defined as follows:

$$p(e)(S)(T) = \forall s.S(s) \Rightarrow \\ \left(\begin{array}{l} trm(op(e))(s \text{ WITH } inp \mapsto o(e)) \\ \wedge\ (\forall s'.(s \text{ WITH } inp \mapsto o(e), s' \text{ WITH } outp \mapsto i(e)) \in prd(op(e)) \Rightarrow T(s'))) \end{array} \right)$$

The *WITH* operator makes the special variables inp and $outp$ of event e correspond to the actual output and input parameters of operation $op(e)$, respectively. Corollary 3.1 of [6] tells us in essence that $\overrightarrow{every}(p)(init)(true)$ characterizes all non-divergent traces of a controller. Thus a controller is consistent if it satisfies the $\overrightarrow{every}$ property.

With this definition, Evans & Treharne [8] define a fixed-point rule for deducing the non-divergence of a controller from some conditions to be met by a CLI:

$$\begin{array}{l} \exists CLI.init \subseteq CLI(n) \\ \wedge\ \forall X.\ (\forall i.X(i)\ \textbf{sat}\ \overrightarrow{every}(p)(CLI(i))(true) \Rightarrow \\ \qquad \forall i.F(X)(i)\ \textbf{sat}\ \overrightarrow{every}(p)(CLI(i))(true)) \\ \Rightarrow \mu(F)(n)\ \textbf{sat}\ \overrightarrow{every}(p)(init)(true) \end{array}$$

with F the CSP controller, n the initial state of the controller and X a function associating an different index for each recursive branch of the controller. **sat** is a satisfaction relation that is defined inductively with respect to each operator of a CSP controller (see e.g. [8, Sect. 4.1] for its definition with a prefixing rule). The fixed-point rule relates the use of a CLI for verifying the divergence-freedom of a controller to uniform properties for CSP controllers. The use of uniform properties for CSP controllers lifts the need for preprocessing as done earlier with the explicit construction of a CLI and it generalizes the parallel composition of CSP controllers. Let P and Q be controllers, S, S', T, T' states, p an event predicate and p', p'' versions of p augmented with additional termination and before-after predicates about external channels of P and Q, respectively:

$$\begin{array}{l} P\ \textbf{sat}\ \overrightarrow{every}(p')(S)(T) \\ \wedge\ Q\ \textbf{sat}\ \overrightarrow{every}(p'')(S')(T') \\ \Rightarrow P \| Q\ \textbf{sat}\ \overrightarrow{every}(p)(S \wedge S')(T \wedge T') \end{array}$$

The "augmented" parts of p' and p'' with respect to p contains an information about data passed by the external channels we can rely on and about data passed to the external channels we must guarantee. This way, when composing two controllers through parallel composition, one can ensure that the "rely-guarantee" contracts match each other and that the whole composition is indeed divergence-free. See [8,6] for more details, with an implementation in PVS.

3 Motivating Example

This section presents an example which arose during the process of assembling already formally specified and proved components. Thus, it is related to one of the stumbling blocks in the development of mandatory certified complex systems: the re-use and assembly of already proved parts of software.

In [1] a convoy, the so-called platoon, of autonomous vehicles was fully specified and validated in the framework of the CSP||B methodology. The behaviour of this system is described *in extenso* in [11] for instance. In the context of this paper we are more concerned with the part of the model limited to a single vehicle.

With respect to [1], a single vehicle, one element of that system, is refined in [12] to obtain a more detailed specification. The resulting formal specification was proved to refine – in the traces/failures model of CSP – the previous CSP||B specification. In [12] the refined specification contains an abstract model of a location component that aims at determining the geographic position of the physical vehicle.

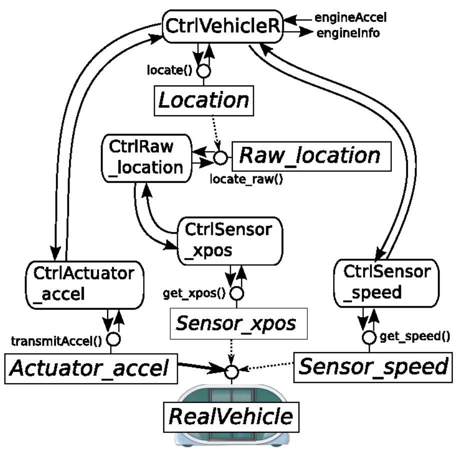

Fig. 1. Simplified view of the CSP||B enhanced vehicle

In the framework of the TACOS project, more concrete specifications of a location component have been independently proposed [2]: one of the introduced safety requirements is that location system would be an assembly of several so-called *raw positioning components* based on different technologies (GPS, Wifi, GSM, Visual sensors,...). Each raw positioning component provides a chronologically ordered set of locations. The sets of all components must be merged. In addition, to (in)validate the provided data, an actual speed and an acceleration can be used. It allows keeping only the possible, i.e. consistent, locations, and removing the inconsistent ones.

Figure 1 presents a simplified CSP||B vehicle model enhanced with the Location component[4]. The conventions are similar to those of Fig. 2, with the plain arrows between CSP processes or between a CSP controller and a B machine being CSP events. In this model, Actuator_accel and Sensor_speed are separate B machines. This is the result of differentiating acceleration values *as they are passed to the engine* and acceleration values *as they have been effectively applied by the engine*.

We want to emphasise the fact that in Fig. 1, some of the CSP controllers share B machines. For example, CtrlVehicleR and CtrlRaw_location share a view on Raw_location. Consequently, the consistency of the whole CSP||B vehicle component risks to be broken because of state sharing. The question we are interested in is: “Is it possible to relax CSP||B restrictions on the architecture of the B part so that we can indeed realise the needed integration?”

4 Recent Works Addressing Sharing in CSP||B and/or B

The sharing of B machines in CSP||B can happen at several levels, in several ways. Figure 2 shows several relevant architectures involving sharing. The “classical” CSP||B diagram is similar to Fig. 2(i) without the shared machine. As no sharing is involved, there is no risk for the invariant of the nonexistent shared machine to be broken, nor

[4] A detailed version of this paper with an appendix depicting a bigger and more complete version of the case study is available at `http://tacos.loria.fr/drupal/?q=node/83`.

for any machine or controller to suffer from interferences from an adjacent controller-machine pair.

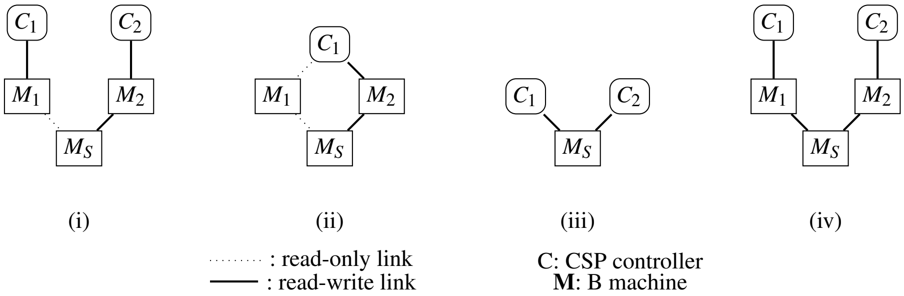

Fig. 2. Several architectures depicting the sharing of B machines

Machine sharing can happen because of sharing by other B machines as in (i), (ii) and (iv) or because of sharing by several controllers as in (iii). Our work is concerned with architectures (i) and (ii), with some considerations about (iv): the novelty of our approach is thus bringing B sharing *at the B level*. This is the reason why we focused primarily on using notions coming from the B setting such as its modularity links. In a nutshell, our approach is about characterizing the links between controllers and machines as *seeing* or *importing* links in the B sense. It then becomes possible to consider the whole CSP part of the system as a single B machine and to use the B constraints upon this "transformed" system to deduce whether the shared B machines of the system can have their invariants broken or not. Let us now compare this approach to similar approaches applied to CSP||B or B alone.

On the one hand, the architecture of Fig. 2(iii) was first introduced in [6], thanks to the use of uniform properties for deciding machine consistency. The reason was that the use of rely-guarantee properties when analyzing the consistency of a controlled machines allowed one controller keeping track of what the other controller could change or not in the machine. Our approach deals mostly with the B part, hence it can be viewed as complementary. This work and ours could thus be used to bring state sharing at every level of the CSP||B formalism.

On the other hand, several works on the B formalism proposed tightened modularity constraints for ensuring the absence of inconsistency or extending the formalism for allowing some useful kinds of sharing. We already mentioned the work of Potet & Rouzaud in Sect. 2.1 that is still situated in the single-writer paradigm. Assuming the CSP controllers can be viewed as a single B entity, the modularity constraints would allow the architectures (i) and (ii) of Fig. 2, because of the clear separation of the seeing (read-only) paths and the importing (read-write) paths. These tightened modularity constraints were integrated quickly into the B commercial tools.

A few works have attempted to deal with the multiple-writer paradigm in B. Boulmé and Potet [13] proposed an approach inspired by a similar technique of Spec#, where a developer can mark at what places the invariant of a shared object (hence, for B, a shared machine) can have its invariant broken. This allows having a broader set of architectures for B but the drawback is a greater number of proof obligations. This approach has no tool support we are aware of.

Büchi and Back [14] proposed changing the B modularity mechanisms to allow for multiple writers in a rely-guarantee fashion. B machines become equipped with contracts, each describing several roles. Each contract corresponds to a way of sharing the machine, with all roles corresponding to a way of invoking the operations of the shared machine. In our opinion, only a combination of CSP with Büchi's B along with the use of uniform properties could deal with the architecture of Fig. 2(iv), because of multiple-writers at the B level and the danger of interferences at the CSP controllers level.

5 Sharing State within CSP||B

Our goal, as exhibited by Sect. 3, is to relax restrictions on the architecture of the B part of a CSP||B model. We shall identify or recall the role played by the current architecture in the verification of CSP||B systems, what would be the consequences of relaxing it and how to avoid the problems caused by this relaxation.

5.1 Consistency Conditions for B and CSP||B

Let us assume here that the initialization is a special kind of operation. In this setting, a B architecture is consistent if the following conditions hold [3,5]:

1. Each machine has its invariant preserved by its operations.
2. Each refinement or implementation can replace the B machine it refines.
3. The whole model respects additional modularity constraints. Among these constraints, such as the absence of cycles in the dependency graph, the most important ones in our context are:
   - A machine can not be imported twice; and
   - The constraint of Theo. 1 (see Sect. 2.1).

The first and the second items are semi-local: the proof obligations correspond to a local reasoning, but the machines can use operations of included or seen machines. It must then be verified that these operations are correctly used: this is done implicitly when operation invocations are expanded into their respective bodies. This ensures that the proof obligation contains a sub-goal for checking that the invoked operation is indeed called within its precondition.

The third point corresponds to a global criterion about the overall architecture of the B project. No double importation and no violation of the constraint of Theo. 1 ensure no invariant breakage and no interference by a machine with a seen machine through another indirect path.

The same kind of conditions is imposed on CSP||B architectures:

- Control loop invariant checking [7] or uniform property verification [6] ensure that a B machine never diverges, i.e. its operations are never called outside their preconditions, through the triggering of its operations by the CSP controller.

- The whole CSP||B model necessarily respects B modularity constraints, because the parts controlled by different CSP controllers never have anything in common.

5.2 B Modular Criteria for the CSP Part of a CSP||B Model

Allowing shared B machines at the B level obviously weakens the second point: is it then possible to express the way the controlled B machines are used by the CSP part in terms of B modularity links and include them in the modularity analysis? The first point could also be impacted by this change: can a controlled machine be interfered by the actions of another controller upon a shared machine, for instance as in Fig. 2(i)?

B Modular Characterization of CSP Control First of all, how can the CSP-controlled operations be characterized in B terms? In [3] Abrial indicates that an operation can be callable, callable in inquiry or not callable. In the first case, such as for an importation link, the called operation can modify the state of the imported machine. In the second case, it can not: it is the case for a seen machine, whose such inquiry operations allow an external machine to observe the state of the seen machine. The third case corresponds to more specific modularity links, such as the **USES** link.

In modular B terms, the control of a B machine can be viewed as a *weakened* **INCLUDES** or **IMPORTS** link: the operations triggered by the CSP part of the system can modify the variables of the controlled machines. A first guideline would thus be that we would consider CSP||B "links" as importation links. We nonetheless can do a finer analysis: it may be the case that a CSP controller never modifies the state of its controlled machine but merely passes around the result of calculations, for instance. We could thus characterize CSP||B links with the following definition:

Definition 1. *If all the operations of a B machine triggered by its CSP controller are inquiry operations in the B sense, then we say that the CSP controller* sees *its controlled B machine. Otherwise, we will say that the CSP controller* imports *its controlled B machine.*

For example, if at least one operation modifies the state of the controlled B machine, then the CSP controller imports its controlled B machine.

Detecting whether an operation is an inquiry operation is rather straightforward: it is defined as being an operation *not* changing the variables of its component [15, Annex E]. Let M be a machine, v any of its variables, op an operation of M, E an expression, S a set, called_op an operation of a machine included in M and P a predicate. We say that op is an inquiry operation if its body contains no substitution of the following shape: $\mathsf{v} := \mathsf{E}$, $\mathsf{v} \leftarrow \mathsf{called\_op}$, $\mathsf{v} :\in \mathsf{S}$ or $\mathsf{v} : (\mathsf{P})$.

Finding if an operation is an inquiry operation can thus be done at the syntactic level, by detecting whether the variables of the machines appear in the left members of the modifying substitutions of the considered operation or not. If all the operations triggered by a controller are inquiry operations, then the control link is a *seeing* link, else it is an *importing* link.

We can now characterize the CSP controls of the B part in terms of the modularity of B. The remaining question is: Can we express the CSP part of a CSP||B system as a B entity? It turns out that expressing a CSP system as a B machine, without loss of semantics, is possible.

Butler [16] proposed a way of translating CSP systems to action systems, which was later adapted to the B method [17]. In essence, the translation matches CSP events with B operations and the result is very close in aspect to what Event-B would look like if expressed with "classical" B. This approach is furthermore supported by the *csp2b* tool, written in MoscowML. The translation keeps the semantics of the CSP operators (sequence, parallel, interleaving) with the additional following constraints: interleaving can only happen at the outermost level and another constraint relevant to the use of so-called "conjoined" B machines, which is a peculiarity of *csp2b* that we do not use. Thus in the end, we now know that viewing as a B entity the CSP part of a CSP||B system is possible.

From CSP to B Modularity We thus know that we are able to translate a CSP system into B. We might stop here and use this translation, with adding what is needed for translating the CSP||B links. We can also notice that the verifications for sharing a B machine are lifted to the architecture of the project. This verification is done through two steps:

- Verifying that the way the variables and operations are used matches the kind of modularity link that is used, for each machine. For instance, verifying that the operations of a seen machine are inquiry operations.
- Verifying that the architecture respects the modularity constraints imposed by the B method, such as the constraint of at the beginning of Sect. 5.2.

Because we characterized the CSP↔B links by means of the **IMPORTS** or **SEES** links depending on what operations the controllers use, we obtain the first step by virtue of construction. We are left with the second step: the content of the B machine does not matter for this step. This means that the content of the CSP system translated into B does not matter either.

Property 1. Let us suppose the CSP part is viewed as a single B machine, and that the links between CSP controllers and B machines are characterized either as **IMPORTS** links or as **SEES** links. If the resulting system respects the modularity constraints of B, then no shared machine in the B part of the system can have its invariant broken.

This property is a direct consequence of lifting all the CSP parts of the system into a B setting: any B architecture that respects the modularity constraints ensures this property.

The process for checking that the B part of a CSP||B system with sharing of B machines is consistent becomes as follows:

1. Characterize the links of each controller to its controlled machine in a B fashion (**IMPORTS** or **SEES**).
2. Represent the whole CSP system (with the CSP controllers) as a single B machine which imports or sees the various controlled machines, depending on how the links have been characterized.

3. Check the resulting pure B architecture with usual B tools.

If the tool checking is successful, then the way the B machine is shared in the whole CSP||B system is consistent. If it fails, then the shared machine(s) face(s) a potential invariant breakage. We shall illustrate this step on an example in Sect. 5.4.

We so far have answered the question about sharing in the sole B context. We shall now provide answers for the problem of divergence-freedom of controllers.

5.3 Ensuring Divergence-freedom in the Presence of Shared B Machines

Let us consider the architecture of Fig. 2(i): the M_S machine is imported by M_2 and seen by M_1, which are themselves imported by their respective controllers. This architecture is sound with respect to the architectural constraints of Sect. 5.2, hence M_S will not have its invariant broken.

Let us now imagine that an operation op_1 of M_1 references some variable of M_S in its precondition, e.g. in the shape of $x_S > 0$. The invariant of M_1 relies thus indirectly upon the strict positivity of x_S. Let us suppose that checking the consistency of $C_1 \| M_1$ does not show any problem. Then, what happens if M_2, because it includes or imports M_S, triggers an operation that makes $x_s = 0$? Then the precondition of op_1 becomes invalid, even though consistency checking did not exhibit the problem. The problem depicted here is typically a problem of non-interference and the consistency checking approach as presented in Sect. 2 is not sufficient.

Fortunately Evans & Treharne [6] encountered a similar problem for related but different reasons. Their first reasoning about composing parallel components involved the reducing of the compounds to a single CSP controller↔B machine compound: the states spaces of the B machine (like Fig. 2(i) without M_S) could be merged together and the controllers assimilated to a single controller with parallel composition. As a side note, this is where parallel composition re-appears at the level of controllers. They reasoned that by doing non-interference checking for separate events and after-states intersection for synchronized events, one can deduce divergence-freedom in the presence of parallel composition of controllers. The rules, translated from the PVS version of [6], are as follows. Let P and Q be CSP controllers and $\Sigma(X)$ the events alphabet of process X:

Definition 2. $if \quad \forall e \in \Sigma(P) \setminus \Sigma(Q)$
$\Rightarrow \forall f \in Sigma(Q)$
$\Rightarrow \forall S, S_1, S_2, (p(S,S_1)(e) \wedge p(S,S_2)(f) \Rightarrow p(S_1,S_2)(f)$
then P does not interfere with the traces of Q, denoted as $non\_interference(p,P,Q)$.

Property 2. if $\quad non\_interference(p,P,Q)$
$\wedge\ non\_interference(p,Q,P)$
$\wedge\ P\ \mathbf{sat}\ \overrightarrow{every}(p)(S)(T)$
$\wedge\ Q\ \mathbf{sat}\ \overrightarrow{every}(p)(S)(T)$
then $P \| Q\ \mathbf{sat}\ \overrightarrow{every}(p)(S)(T)$

There is an additional rule for the intersection of after-states of synchronized events that we do not show here, because we only deal with machine channels, hence events performed independently by each controller.

It turns out that this property can be simplified in our architectural case: the non-interference property is used in a case similar with Fig. 2(iii) because the controllers both "import" the shared machine, hence can interfere with one another. In our architectural case, we know that the shared machine is effectively imported only by one controller, because of the B rule stating that a machine can only be imported once. Hence we know that this shared machine will be unaffected by all other controllers: they will only ultimately be allowed to refer to the shared machine through **SEES** links, hence they can never modify the shared machine. We thus integrate this specificity in Prop. (2), leading to:

Property 3. if P is a controller that ends up importing a shared machine, as in the architecture of Fig. 2(i), and $non\_interference(p,P,Q)$
$\wedge\ P\ \mathbf{sat}\ \overrightarrow{every}(p)(S)(T)$
$\wedge\ Q\ \mathbf{sat}\ \overrightarrow{every}(p)(S)(T)$
then $P||Q\ \mathbf{sat}\ \overrightarrow{every}(p)(S)(T)$

As the non-interference property is trivially verified for Q upon P thanks to the knowledge about the architecture of the system, we simply removed it. The other non-interference properties must be kept, because as P imports the shared machine, it can still have an effect on the other controllers that see the shared machine.

We now are able to answer to the questions of this section. If a CSP||B system with machine sharing in the B part meets the following requirements:

- The CSP system viewed as a B entity together with the B part respects Prop. 1
- The controllers, at least those that involve shared machines, respect Prop. 3

then this CSP||B system is consistent for the parts involved in the sharing of B machines. The rest of the system can be verified e.g. with the techniques of [6]. The next section illustrates these results on our running example.

5.4 Translating the Simplified Vehicle System into a B Architecture

Let us consider again Fig. 1. Let us denote M the B entity corresponding to the CSP processes (or controllers): CtrlVehicleR, CtrlActuator_Accel, CtrlSensor_Speed, CtrlRaw_location and CtrlSensor_xpos. Although there is no direct link between CtrlVehicleR and CtrlRaw_location, they are still executed in parallel and could cause invariant breakage in a commonly shared B machine.

Now let us write the *sees* and *imports* sets for calculating whether the architecture respects the condition stated above. We kept the names of the differentiated CSP controllers/processes instead of using M so that the reader can more easily follow the steps with respect to Fig. 1. The controller↔machine links are importation links because the machines are modified, as they are used for backing up the passed value in a log. With that in mind, the *sees* relationship and the *imports* relationship are as follows. Note that we omitted the reflexive part of the set, such as Sensor_xpos $\mapsto$ Sensor_xpos, etc:

$$
\begin{array}{c}
sees = \\
\left\{\begin{array}{lcl}
\mathsf{Sensor\_xpos} & \mapsto & \mathsf{RealVehicle} \\
\mathsf{Sensor\_speed} & \mapsto & \mathsf{RealVehicle} \\
\mathsf{Location} & \mapsto & \mathsf{Raw\_location}
\end{array}\right\} \\
imports = \\
\left\{\begin{array}{lcl}
\mathsf{Actuator\_accel} & \mapsto & \mathsf{RealVehicle} \\
\mathsf{CtrlActuator\_accel} & \mapsto & \mathsf{Actuator\_accel} \\
\mathsf{CtrlSensor\_speed} & \mapsto & \mathsf{Sensor\_speed} \\
\mathsf{CtrlSensor\_xpos} & \mapsto & \mathsf{Sensor\_xpos} \\
\mathsf{CtrlRaw\_location} & \mapsto & \mathsf{Raw\_location} \\
\mathsf{CtrlVehicleR} & \mapsto & \mathsf{Location}
\end{array}\right\}
\end{array}
\quad
\begin{array}{c}
((sees \cup imports)^{+})^{*} = \\
\left\{\begin{array}{lcl}
\mathsf{Sensor\_xpos} & \mapsto & \mathsf{RealVehicle} \\
\mathsf{Sensor\_speed} & \mapsto & \mathsf{RealVehicle} \\
\mathsf{Location} & \mapsto & \mathsf{Raw\_location} \\
\mathsf{Actuator\_accel} & \mapsto & \mathsf{RealVehicle} \\
\mathsf{CtrlActuator\_accel} & \mapsto & \mathsf{Actuator\_accel} \\
\mathsf{CtrlSensor\_speed} & \mapsto & \mathsf{Sensor\_speed} \\
\mathsf{CtrlSensor\_xpos} & \mapsto & \mathsf{Sensor\_xpos} \\
\mathsf{CtrlRaw\_location} & \mapsto & \mathsf{Raw\_location} \\
\mathsf{CtrlVehicleR} & \mapsto & \mathsf{Location}
\end{array}
\begin{array}{lcl}
\mathsf{CtrlSensor\_xpos} & \mapsto & \mathsf{RealVehicle} \\
\mathsf{CtrlSensor\_speed} & \mapsto & \mathsf{RealVehicle} \\
\mathsf{CtrlVehicleR} & \mapsto & \mathsf{Raw\_location} \\
\mathsf{CtrlActuator\_accel} & \mapsto & \mathsf{RealVehicle} \\
\mathsf{Sensor\_xpos} & \mapsto & \mathsf{Sensor\_xpos} \\
\dots & & \dots\ \dots
\end{array}\right\}
\end{array}
$$

We finally calculate the possibly, and indirectly, modified machines:

$$
((sees \cup imports)^{+})^{*}; imports =
\left\{\begin{array}{lcl}
\mathsf{Actuator\_accel} & \mapsto & \mathsf{RealVehicle} \\
\mathsf{CtrlActuator\_accel} & \mapsto & \mathsf{Actuator\_accel} \\
\mathsf{CtrlSensor\_speed} & \mapsto & \mathsf{Sensor\_speed} \\
\mathsf{CtrlSensor\_xpos} & \mapsto & \mathsf{Sensor\_xpos} \\
\mathsf{CtrlRaw\_location} & \mapsto & \mathsf{Raw\_location} \\
\mathsf{CtrlVehicleR} & \mapsto & \mathsf{Location} \\
 & & \\
\mathsf{CtrlActuator\_accel} & \mapsto & \mathsf{RealVehicle}
\end{array}\right\}
$$

Now after having rewritten the CSP controllers or processes into M we obtain:

$$
\begin{array}{c}
sees = \\
\left\{\begin{array}{lcl}
\mathsf{Sensor\_xpos} & \mapsto & \mathsf{RealVehicle} \\
\mathsf{Sensor\_speed} & \mapsto & \mathsf{RealVehicle} \\
\mathsf{Location} & \mapsto & \mathsf{Raw\_location}
\end{array}\right\}
\end{array}
\quad
\begin{array}{c}
((sees \cup imports)^{+})^{*}; imports = \\
\left\{\begin{array}{lcl}
\mathsf{Actuator\_accel} & \mapsto & \mathsf{RealVehicle} \\
\mathsf{M} & \mapsto & \mathsf{Actuator\_accel} \\
\mathsf{M} & \mapsto & \mathsf{Sensor\_speed} \\
\mathsf{M} & \mapsto & \mathsf{Sensor\_xpos} \\
\mathsf{M} & \mapsto & \mathsf{Raw\_location} \\
\mathsf{M} & \mapsto & \mathsf{Location}
\end{array}\right\}
\end{array}
$$

Note that M will never be a target, because the whole CSP part will always be a source of inclusion/sight towards B machines. The intersection of the relations in the tables above is obviously empty, hence the architectural criterion is respected.

The divergence-freedom of the controlled machines is also respected. Although the code of the machines is not shown here, it is very simple as we do not make strong assumptions about the passed values at the moment. The various preconditions of the machines are thus merely for typing the variables: as it is the "maximal" property possible for a variable, the absence of interference is trivially verified.

5.5 Discussion: Other Architectural Patterns

The solution for introducing shared B machines in a CSP||B system also gives clues about other kinds of architectural evolutions for a CSP||B system. The "one machine-several controllers" as in Fig. 2(iii) is already handled by the consistency definition of Evans & Treharne [6].

The "one controller-several machines" case as of Fig.2(ii) is also conjectured to be solved by our approach. Assuming that the controller does not contain any parallel composition, as is the case usually for CSP controllers, then there is no interference problem. Hence the problem here is strictly reduced to the verification of modular constraints. In case both controlled machines are imported by the CSP controller, our approach does not allow to decide the (in)consistency of the shared machine.

We are left with the case of Fig.2(iii) when modifications happen for all links. In that case, the base assumptions of B modularity are obviously not met, hence apart from the full use of consistency checking techniques from [6], one would have to use an extension of B allowing such modularity links. From Sect. 4 we can surmise that the “invariant ownership” approach of [13] or the “rely-guarantee” approach of [14] would fit. Given that Boulmé concludes that [13, conclusion, third paragraph] the rely-guarantee approach is more modular, we suggest that using Büchi’s extension of B as a replacement for classical B would bring what is needed for such an architectural case. As this extension impacts mostly the modularity of B and not its core (set theory and substitutions), we think the changes needed at the level of CSP||B would be minor.

6 Conclusion

When integrating independent work on a B model of a location component [2] in our whole CSP||B system [1], we were faced with the violation of CSP||B architectural constraints by the newly introduced location component. The CSP||B formalism indeed requires that the controlled B components do not share states.

The solution we propose for allowing *some* architectures with state sharing involves the verification that the shared machine has not its invariant broken, and that the introduction of sharing does not disturb the components. As the first verification is rooted into B semantics, we proposed an idea based on the fact that a CSP system can be viewed as a B machine. We thus were left with characterizing the links between CSP controllers and B machines as B modularity links. The verification could thus be reduced to the checking that B modularity constraints are satisfied.

The second verification involved problems of interference between controllers. We re-used the solutions proposed by Evans & Treharne [6] for verifying the non-interference of controllers. We adapted them with the additional knowledge given by modularity links at the B level to deduce that some non-interference properties were naturally drawn from the modularity links.

Our solution allows the relaxation of some constraints upon B machines in a CSP||B system. From there, we conjecture that most architectural patterns can be solved with a combination of our solution and the consistency checking rules of [6]. We think at this point that, for addressing the multiple-writers problem at both the level of CSP||B and B, one would need using another extension of B allowing such a paradigm, such as a version of B extended with rely-guarantee contracts [14].

Longer-term perspectives include the study of CSP||B component refinements adapted to our problem. Preliminary studies of recent advances in this domain [18] imply that the kind of refinement we seek would be different because of a more complex evolution of the B part through the design. Other interesting perspectives would involve the adaptation of the consistency rules of [6] from PVS to a library for the B method in Coq [19], as the affinity of Coq with fixed-point reasoning could help in the verification of uniform properties.

References

1. Colin, S., Lanoix, A., Kouchnarenko, O., Souquières, J.: Towards Validating a Platoon of Cristal Vehicles using CSP||B. In Meseguer, J., Rosu, G., eds.: 12th International Conference on Algebraic Methodology and Software Technology (AMAST 2008). Number 5140 in LNCS, Springer-Verlag (2008) 139–144
2. Matoussi, A., Laleau, R., Petit, D.: Improving traceability between KAOS requirements models and B specifications. In: 7th IEEE International Conference on Software Engineering and Formal Methods (SEFM'09), Hanoi, Vietnam (2009) in submission (tech. report: `http://tacos.loria.fr/drupal/?q=node/32`).
3. Abrial, J.R.: The B Book - Assigning Programs to Meanings. Cambridge University Press (1996)
4. Potet, M.L., Rouzaud, Y.: Composition and refinement in the B method. In: B'98 : The 2nd International B Conference. (1998) 46–65
5. Rouzaud, Y.: Interpreting the B-method in the refinement calculus. In: Proceedings of FM'99: World Congress on Formal Methods. Volume 1709 of LNCS., Springer-Verlag (1999) 411–430
6. Evans, N., Treharne, H.: Interactive tool support for CSP || B consistency checking. Formal Aspects of Computing **19**(3) (2007) 277–302
7. Schneider, S.A., Treharne, H.E.: CSP theorems for communicating B machines. Formal Aspects of Computing, Special issue of IFM'04 (2005)
8. Evans, N., Treharne, H.E.: Investigating a file transfer protocol using CSP and B. Software and Systems Modelling Journal **4** (2005) 258–276
9. Schneider, S., Cavalcanti, A., Treharne, H., Woodcock, J.: A layered behavioural model of platelets. In: 11th IEEE International Conference on Engieerging of Complex Computer Systems, ICECCS. (2006)
10. Evans, N., Treharne, H.E.: Linking semantic models to support CSP||B consistency checking. In: AVOCS'05. (2005)
11. Lanoix, A.: Event-B specification of a situated multi-agent system: Study of a platoon of vehicles. In: 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE), IEEE Computer Society (2008) 297–304
12. Colin, S., Lanoix, A., Kouchnarenko, O., Souquières, J.: Using CSP||B Components: Application to a Platoon of Vehicles. In: 13th International ERCIM Wokshop on Formal Methods for Industrial Critical Systems (FMICS 2008). Number 5596 in LNCS, Springer-Verlag (2009) 103–118
13. Boulmé, S., Potet, M.L.: Interpreting invariant composition in the B method using the spec# ownership relation: A way to explain and relax B restrictions. In: The 7th International B Conference. Volume 4355 of LNCS., Springer (2007) 4–18
14. Büchi, M., Back, R.: Compositional symmetric sharing in B. In: Proceedings of FM'99: World Congress on Formal Methods. Volume 1709 of LNCS., Springer-Verlag (1999) 431–451
15. Clearsy: B language reference manual. v1.8.6 edn. (2007) `http://www.atelierb.eu`.
16. Butler, M.J.: A CSP Approach To Action Systems. PhD thesis, Oxford University (1992)
17. Butler, M.J.: CSP2B : A practical approach to combining CSP and B. In: Proceedings of FM'99: World Congress on Formal Methods. Volume 1709 of LNCS., Springer-Verlag (1999) 490–508
18. Schneider, S., Treharne, H.: Changing system interfaces consistently: A new refinement strategy for CSP||B. In: Integrated Formal Methods, 7th Int. Conf. IFM 2009, Düsseldorf, Germany, February 16-19, 2009. Volume 5423 of LNCS., Springer (2009) 103–117
19. Colin, S., Mariano, G.: BiCoax, a proof tool traceable to the BBook. In: From Research to Teaching Formal Methods - The B Method (TFM B'2009). (2009)