Relaxing B Sharing Restrictions within CSP||B

Arnaud Lanoix, Olga Kouchnarenko, Samuel Colin, Vincent Poirriez

Relaxing B Sharing Restrictions within CSP‖B*

Arnaud Lanoix[1], Olga Kouchnarenko[2], Samuel Colin[3], and Vincent Poirriez[4]

[1] LINA CNRS and Nantes University, Nantes, France
arnaud.lanoix@univ-nantes.fr
[2] FEMTO-ST CNRS and University of Franche-Comté, Besançon, France
olga.kouchnarenko@univ-fcomte.fr
[3] SafeRiver, France
scolin@hivernal.org
[4] LAMIH CNRS and University Lille Nord de France, Valenciennes, France
vincent.poirriez@univ-valenciennes.fr

Abstract. This paper addresses the issue of state sharing in CSP‖B specifications: B machines controlled by various CSP parts are supposed not to refer to, share or modify the same state space. However, some kinds of B state sharing can be allowed without creating inconsistencies in CSP‖B specifications. To achieve this, we present a B-based solution for allowing architectures with B state sharing in the CSP‖B components. We show that the inconsistencies in state sharing can be identified by translating the CSP controllers into B specifications and then using a more refined consistency checking process. We also hint at possible extensions towards other CSP‖B architectural patterns with various types of sub-components sharing.
Keywords: CSP‖B, sharing, architecture, consistency, rely-guarantee

1 Introduction

In this work we address the question of how to safely reuse already-developed B component 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 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, like in the location component, is not valid at the CSP‖B level. In fact, such an architecture goes against the general (and well-known) "one controller≡one machine" CSP‖B constraint.

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

Moreover, although more recent results allow several controllers to share a B machine, like in [3], they do not permit to deal with our case study.

Several relevant architectures involve B state sharing which can happen because of sharing a B machine by other B machines. 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 characterising 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 decide whether the shared B machines of the system can have their invariants broken or not.

Unlike [3,4], the novelty of our approach is thus bringing B sharing *to the B level*. Indeed, in [4], Evans et.al used CSP controllers "augmented" with a B part to perform automatic consistency and non-discrimination checks of CSP‖B models. In that work, determining which parts of a CSP‖B analysis can be handled within the B method has been left aside as a future work direction. The approach advocated in the present paper deals mostly with the B part, hence it can be viewed as complementary. Those works and ours could thus be used together to bring state sharing at every level of the CSP‖B formalism. More precisely, we show how to use the B modularity constraints to allow CSP‖B models with multiple controllers for a B machine or with a single controller for multiple, and possibly shared, B machines. We then propose a refined consistency checking of CSP‖B based on such architectural patterns.

Layout of the paper. Before introducing a platoon example and a part of its modelling in Section 3, we present the necessary formalisms, concepts and tools in Sect. 2. Our main contributions are in Sect. 4 and 5. We propose 1) a method—based on the B modularity—for detecting inconsistent CSP‖B architectures, and 2) a refinement of CSP‖B consistency check requirements based on architectural patterns. In addition, we propose extensions for addressing the verification of more complex cases. Finally, conclusions and assessments are drawn in Sect. 6, combined with related work on state sharing in CSP‖B and B.

2 Concepts and Tools for CSP‖B Components

The B machines specifying components are open modules which interact by the authorised 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.

2.1 B Machines

B is a formal software development method used to model and reason about systems [5]. The B method has proved its strength in industry with the development

of complex real-life applications such as the Roissy VAL [6]. The principle behind building a B model is the expression of system properties which are always true after each evolution step of the model, the evolution being specified by the B operations. The verification of a model correctness is thus akin to verifying the preservation of these properties, no matter which step of evolution the system takes.

The B method is based on first-order logic, set theory and relations. A strength of the B method is its stepwise refinement feature: each refinement makes a model more deterministic and also more precise by introducing programming language-like features, until the code of the operations can actually be implemented in a programming language.

Let us assume here that the initialisation is a special kind of operation. In this setting, a B architecture is *consistent* if the following conditions hold [5,7]:

- Each machine has its invariant preserved by its operations, i.e. the model is *consistent*.
- Each refinement or implementation can replace the B machine it refines.

Both items above 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.

Support tools such as B4free (`http://www.b4free.com`) or AtelierB (`http://www.atelierb.eu`) automatically generate Proof Obligations (POs) to ensure the consistency [5]. Some of them are "obvious" POs which are automatically discharged whereas the normal POs have to be proved interactively if it was not done fully automatically.

Modularity in B The B project architecture can be handled through some specific clauses **SEES**, **INCLUDES** and **USES** that allow a machine to list its seen machines, included machines or used machines, respectively. The **IMPORTS** clause corresponds to **INCLUDES** for an implementation model. A B architecture must respect some *modularity* constraints. For instance, one machine cannot end up being included or imported twice by two different inclusion paths, as this could break the invariant. In [8] the modularity constraints in [5] have been proved to be not strong enough, because intermediate **SEES** links could hide the fact that a machine could be modified through refinement. In [7], a modularity constraint to ensure no invariant breakage and no interference by a machine with a seen machine through another indirect path, is given:

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

with *sees* being 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 directly "sees" 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. 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.

When taking into account all implicit hypotheses about B modularity [7], the formula can actually be simplified into the following shape: $can\_alter \cap sees = \emptyset$. We pointed out this modularity constraint because of the role it plays in our contribution in Sect. 4.

2.2 Communicating Sequential Processes (CSP)

CSP allows the description of entities, called processes, which exist independently but may communicate with each other. Thanks to dedicated operators it is possible to describe a set of processes as a single process, making CSP an ideal formalism for building a hierarchical composition of components. CSP is supported by the FDR2 model checker (`http://www.fsel.com`). This tool is based on the generation of all the possible states of a model and the verification of these states against a desired property.

The denotational semantics of CSP is based on the observation of process behaviours. Three kinds of behaviours [9] are observed and well suited to express the properties:

- traces, i.e. finite sequences of events, for safety properties;
- stable failures, i.e. traces augmented with a set of unperformable events at the end thereof, for liveness properties and deadlock-freedom;
- failures/divergences, i.e. stable failures augmented with traces ending in an infinite loop of internal events, for livelock-freedom.

Each kind of behaviours gives rise to a notion of process refinement defining a particular semantical framework [9].

2.3 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 [3,10] and the abundant CSP‖B literature referenced therein; for case studies, see for example [11,12].

Specifying CSP controllers In CSP‖B architecture (as depicted Fig. 1), 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:

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

The process $\mathsf{c} \ ? \ \mathsf{x} \ ! \ \mathsf{v} \rightarrow \mathsf{P}$ can accept input x and output v along a communication channel c. Having accepted x, it behaves as P.

Machine channels are introduced in CSP controllers to provide the means for controllers to synchronise with the B machine: for each B operation $x \leftarrow ope(v)$, there can be a channel ope ! v ? 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.

Remark 1. CSP‖B components must respect the "one controller≡one machine" constraint (as shown in Fig. 1): controlled B machines are not allowed to share states, i.e. they cannot *see* or *import* the same machines. Then, the CSP‖B model necessarily respects the B modularity constraints (Theo 1, Sect. 2.1).

The behaviour of a guarded process b & P depends on the evaluation of the boolean condition b: if it is true, it behaves as P, otherwise it is unable to perform any events. In some works (e.g. [10]), the notion of *blocking assertion* is defined by using a guarded process on the inputs of a channel to restrict these inputs: c ? x & E(x) → P.

The external choice P1 □ P2 is initially prepared to behave either as P1 or as P2, with the choice made on the occurrence of the first event. The conditional choice **if** b **then** P1 **else** P2 behaves as P1 or P2 depending on b. Finally, S(p) expresses a recursive call. Finally, in addition to the expression of simple processes, CSP provides parallel composition operators to combine them.

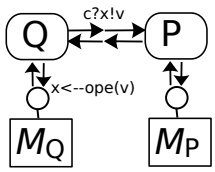


Fig. 1. CSP‖B components

Verifying CSP‖B components The main problem with combined specifications is their *consistency*: CSP and B parts should not be contradictory. Let us assume a CSP‖B compound $(P\|M_P)$. The verification process to ensure the consistency of $(P\|M_P)$ consists in verifying the following conditions [10]:

1. Check the *consistency* of M_P with B4Free or Atelier-B for instance,
2. Check the *deadlock-freedom* (in the stable-failures model) and *divergence-freedom* of P with FDR2,
3. Check the *divergence-freedom* of $(P\|M_P)$ (see below),
4. By way of [10, Theorem 5.9] and the fact that P is deadlock-free, the *deadlock-freedom* of $(P\|M_P)$ in the stable failures model is deduced.

The given results are also generalised in [10] to a collection of B machine-CSP process couples. The whole CSP‖B architecture must also respect the sharing constraint recalled Remark 1.

Ensuring the divergence-freedom of CSP‖B components Originally, the technique for ensuring the *divergence-freedom* of a controlled machine $(P\|M_P)$

involved the stating of a Control Loop Invariant (CLI) and its verification [13,14]. Fortunately, the above technique has evolved into a more general and less cumbersome one. Evans & Treharne [11] have defined a fixed-point rule for deducing the non-divergence of a controlled machine $(P \| M_P)$.

To sum up, the fixed-point rule procedure is based on the satisfaction by the controller P of a uniform property $\overrightarrow{every}(p)(S)(T)$, where p is an event predicate and S, T are states (e.g. predicates expressed in the B set theory): P **sat** $\overrightarrow{every}(p)(S)(T)$. See [11,3] for more details, with a PVS implementation.

That 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 generalises the parallel composition of CSP controllers.

In [3], the authors deduced the divergence-freedom of $P \| Q$ by verifying the *non-interference*, i.e. a property which expresses that P does not interfere with the traces of Q, denoted as $non\_interference(p, P, Q)$. Then, they deduced:

Property 1.

$$\text{If } \left\{ \begin{array}{l} non\_interference(p, P, Q) \\ \wedge\ non\_interference(p, Q, P) \\ \wedge\ P \textbf{ sat } \overrightarrow{every}(p)(S)(T) \\ \wedge\ Q \textbf{ sat } \overrightarrow{every}(p)(S)(T) \end{array} \right\} \text{ then } P \| Q \textbf{ sat } \overrightarrow{every}(p)(S)(T)$$

3 Motivating Case Study

This section presents an example which arose during the process of assembling already formally specified and proved components. In [1] a convoy, the so-called platoon, of autonomous vehicles (depicted in Fig. 2) was fully specified and validated in the framework of the CSP$\|$B methodology. The behaviour of this system is described *in extenso* in [15]. In the context of this paper we are more concerned with the part of the model limited to a single vehicle. Figure 3 illustrates a single vehicle, one element of the platoon. Its formal study can be found in [1].

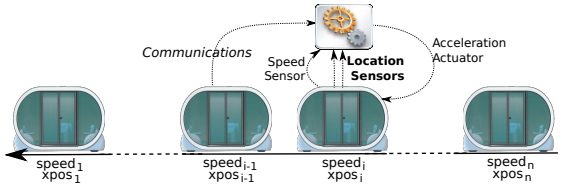


Fig. 2. A platoon of autonomous vehicles as a multi-agent system

In figures the conventions are as follows: the rounded boxes depict CSP controllers, whereas the others show B machines, with the plain arrows between CSP processes or between a CSP controller and a B machine being read-write links, and dotted arrows being read-only links.

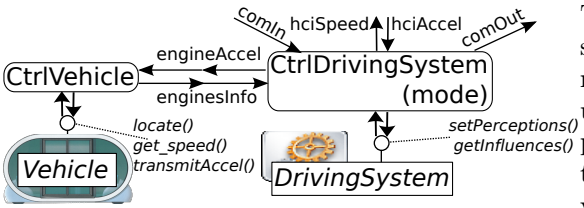


Fig. 3. Abstract CSP‖B vehicle

This first CSP‖B specification was refined in [16]. The resulting more detailed specification was proved to refine – in the traces/failures model of CSP – the previous specification. In [16] the refined specification involves several controllers (instead of the only CtrlVehicle controller) equipped with B machines. It also contains an abstract model of a location component answering the locate() B method by determining the geographic position of the physical vehicle.

In the framework of the TACOS project, more concrete B specifications of the location component have been independently proposed in [2]: an enhanced realistic pure B model of the vehicle (with focus on the location problem) was derived from the requirements specified using the KAOS method [17].

One of the introduced safety requirements is that location sensors would be an assembly of several so-called *raw positioning components* based on different technologies (GPS, Wifi, GSM, Visual sensors,...). Each raw positioning sensor 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 acceleration can be used. It allows keeping only the possible, i.e. consistent, locations, and removing the inconsistent ones.

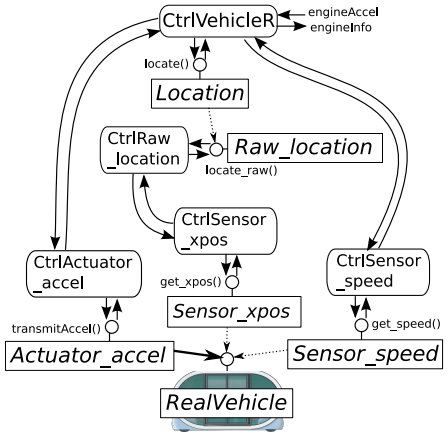


Fig. 4. Enhanced CSP‖B vehicle

Figure 4 displays a simplified CSP‖B vehicle model enhanced with the Location component[5]. 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

[5] 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`.

acceleration values *as they have been effectively applied by the engine*. We want to emphasise the fact that in Fig. 4, 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 B-based State Sharing within CSP‖B

As recalled in Remark 1 (and in Fig. 1), a CSP‖B architecture disallowed any sharing of B machines. This way, there is no risk for the invariant of the nonexistent shared machine to be broken, nor for any machine or controller to suffer from interferences from an adjacent controller-machine pair. However, Figure 5 shows several relevant architectures involving B state sharing. Machine sharing can happen because of sharing by other B machines as in (a), (b) and (d) or because of sharing by several controllers as in (c).

Our goal, as exhibited in Sect. 3, is to relax restrictions on the architecture of the B part of a CSP‖B model. In this section, we show that it is possible to express the way the controlled B machines are used by the CSP part in terms of B modularity links, and to include them in the B modularity checking, to allow B state sharing in CSP‖B.

More precisely, we are concerned with architectures (a) and (b), with some considerations about (d): the novelty of our approach is thus bringing B sharing *to 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 characterising 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 decide whether the shared B machines of the system can have their invariants broken or not.

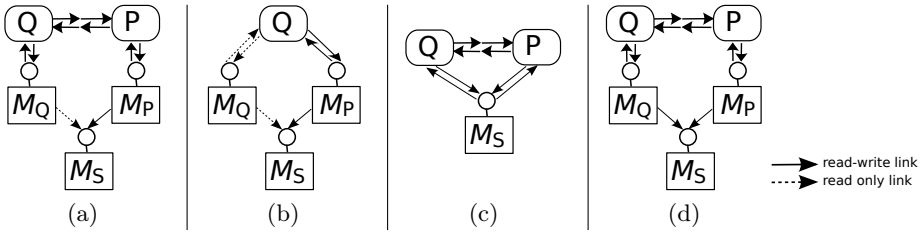


Fig. 5. Several architectures depicting the sharing of B machines

4.1 B Modular Characterisation of CSP Control

We want to characterise in B terms, the *machine channels*, i.e. the CSP-controlled operations. In [5] 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 cannot: 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 CSP control of a B machine can be viewed as a kind of **INCLUDES** or **IMPORTS** links: 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 **IMPORTS** 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 characterise 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.*

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 [18, Annex E]. 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.

This way we can characterise the CSP controls of the B part in terms of the modularity of B. Then, we want to express the CSP part of a CSP‖B system as a B entity, to check the B modularity constraints on the whole CSP‖B system.

4.2 From CSP to B Modularity

It is well-known that a CSP system can be translated into B using results in [19,20]. We might stop here and use this translation, with adding what is needed for translating the CSP‖B links. Instead we go further by exploiting the fact that the verifications to correctly share a B machine are lifted to the architecture of the project. Indeed, these verifications can be done through two B-based 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 in Sect. 4.1.

Because we characterised 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 2. Let the CSP part be represented by a single B machine, and the links between CSP controllers and B machines be characterised 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.

Proof. (Sketch) (i) Let us assume that the translation from CSP into B is correct. It is based on the results in [20]. (ii) The interactions between CSP and B parts can be characterised in terms of the B modularity (see Sect. 4.1).
Consequently, if the whole system expressed in B thanks to (ii) satisfies the modularity constraints of B given by Theo 1, Sect. 2.1 then, by (i), the CSP‖B system also satisfies the modularity constraints, and no shared B machine has its invariant broken. Obviously, the last point only concerns the B part. □

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.

Thanks to our proposals, the process for checking that the B part of a CSP‖B system with sharing of B machines is consistent becomes as follows:

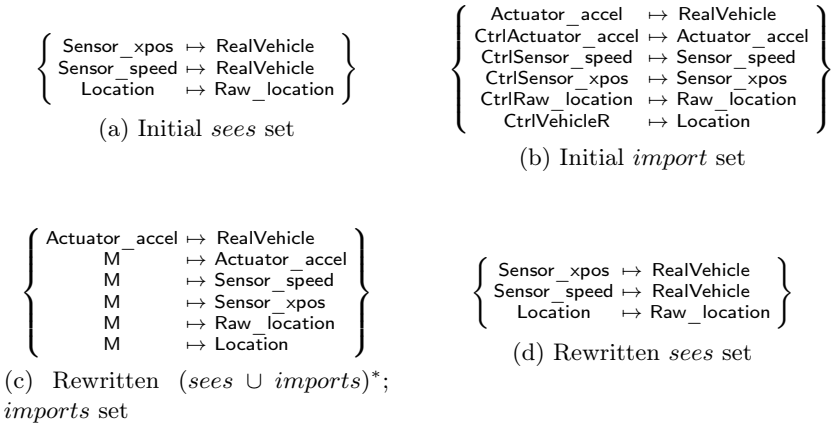
1. Characterise the links of each controller to its controlled machine in a B fashion (**IMPORTS** or **SEES**).
2. Represent the whole CSP system (with the CSP controllers) as a single B machine (using *csp2b* for instance [19,20]) which imports or sees the various controlled machines, depending on how the links have been characterised.
3. Check the resulting pure B architecture with usual B tools, B4free or Atelier-B for instance.

Notice that Property2 is a sufficient but not necessary condition. 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 machines face a potential invariant breakage. The example in the next section illustrates this step.

4.3 Application to the Vehicle System

Let us consider again Fig. 4. Let M be 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. Let us analyse this.

Let us write the *sees* and *imports* sets depicted by Figs 6a and 6b for calculating whether the architecture respects the B modularity constraints. We kept



(a) Initial *sees* set

(b) Initial *import* set

(c) Rewritten $(sees \cup imports)^*$; *imports* set

(d) Rewritten *sees* set

Fig. 6. *sees*, *imports* and $(sees \cup imports)^*; imports$ sets

the names of the differentiated CSP controllers/processes with respect to Fig. 4 instead of using M. 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. Now after having rewritten the CSP controllers or processes into M, the final $(sees \cup imports)^*; imports$ set which contains the possibly, and indirectly, modified machines is given Fig. 6c. 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 Figs 6d and 6c is empty, hence the architectural B criterion (Sect. 2.1 and 4.1) is satisfied.

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.

5 Ensuring Divergence-freedom of Shared B Machines

Control loop invariant checking [14,10] or uniform property verification [3] ensure that a controlled B machine never diverges, i.e. its operations are never called outside their preconditions, through the triggering of its operations by the CSP controller.

Let us consider the architecture of Fig. 5(a): the M_S machine is imported by M_P and seen by M_Q, which are themselves imported by their respective controllers P and Q. This architecture is sound with respect to the architectural constraints of Sect. 4.1, hence M_S will not have its invariant broken.

Let us now imagine that an operation ope_q of M_Q references some variable of M_S in its precondition, e.g. in the shape of $x_S > 0$. The invariant of M_Q relies thus indirectly upon the strict positivity of x_S. Let us suppose that checking the

consistency of $Q \| M_Q$ does not show any problem. Then, what happens if M_P, because it includes or imports M_S, triggers an operation that makes $x_s = 0$? Then the precondition of ope_q 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.

Let us notice that in [3] the authors encountered a similar problem for related but different reasons. Their non-interference Property 1 recalled Sec. 2 is used in a case similar to the architectural case illustrated by Fig. 5(c) because the both controllers "import" the shared machine, hence can interfere with each another.

Fortunately, it turns out that Property 1 can be simplified in our architectural case depicted Fig. 5(a). Indeed, 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 Property 1, leading to:

Property 3. If P is a controller that ends up importing a shared machine (Fig. 5(a)), and

$$\left\{ \begin{array}{l} non\_interference(p, P, Q) \\ \wedge\ P \textbf{ sat } \overrightarrow{every}(p)(S)(T) \\ \wedge\ Q \textbf{ sat } \overrightarrow{every}(p)(S)(T) \end{array} \right\} \text{ then } P \| Q \textbf{ sat } \overrightarrow{every}(p)(S)(T)$$

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

Proof. (Sketch) Let assume without loss of generality that the whole CSP‖B system satisfies the modularity constraints (Sect. 4.1). As a consequence, in our architectural case only P can write into M_S. Hence Q (or other seeing controllers) can only use non-modifying operations of M_S. As a result, Q does not interfere with the P behaviour. This can be shown (i) by induction on the traces tr—universally specified in $every(p)(S)(T)(tr)$—of invocations by P of operations of the controlled B machines M_P and M_S, and (ii) by analysis of the effect of M_S operations called by Q via M_Q: as operations are non-modifying there is no interference in this case. On the other hand, because of the $non\_interference(p, P, Q)$ hypothesis, P does not interfere with the Q behaviour, and we are done. □

Thanks to our proposals, the restriction on state sharing in CSP‖B can be relaxed as follows. 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 Property 2 (as presented in the previous section)

– The controllers, at least those that involve shared machines, respect Property 3

then the CSP‖B system is consistent for the parts sharing B machines. The rest of the system can be verified e.g. with the techniques of [3].

Discussion about 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. 5(c) is already handled by the consistency definition in [3]. The "one controller-several machines" case illustrated by Fig. 5(b) is 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 B 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. 5(c) when modifications happen for all links. In that case, the basic assumptions of B modularity are obviously not met, hence apart from the full use of consistency checking techniques from [3], one would have to use an extension of B allowing such modularity links. We can surmise that the "invariant ownership" approach of [21] or the "rely-guarantee" approach of [22] would fit. Given that Boulmé concludes that [21, 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

This paper proposed a B-based solution for allowing architectures with B state sharing in the CSP‖B components. The proposal involved the verification that the shared B machine has not its invariant broken, and that the introduction of sharing does not disturb the components. As the first verification is rooted to B semantics, we proposed a verification methodology based on the fact that the CSP parts of the system can be viewed as a single B machine. We thus were left with characterising the links between CSP controllers and B machines as B modularity links. We have shown that the verification could thus be reduced to check that the B modularity constraints are satisfied.

The second verification involved problems of interference between controllers. We adapted and simplified the solution proposed by Evans & Treharne [3] for verifying the non-interference of controllers. We exploit the additional knowledge given by modularity links at the B level to naturally deduce non-interference properties from the modularity links.

Related Work Addressing Sharing in B and CSP‖B Let us now compare this approach to similar approaches applied to CSP‖B or B alone (see Sect. 4).

On the one hand, the architecture of Fig. 5(c) was first introduced in [3], thanks to the use of uniform properties for deciding machine consistency. The reason was that the use of rely-guarantee properties when analysing the consistency of a controlled machine allowed one controller keeping track of what the other controller could change or not in the machine. In [4], Evans et.al used CSP controllers "augmented" with a B part to perform automatic consistency and non-discrimination checks of CSP‖B models of information systems. Our approach deals mostly with the B part, hence it can be viewed as complementary. Those works and ours could thus be used together 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. The already mentioned in Sect. 2.1 works in [8,7] are 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 (a) and (b) of Fig. 5, because of the clear separation of the seeing (read-only) paths and the importing (read-write) paths. These tightened modularity constraints were quickly integrated into the B commercial tools.

A few works have attempted to deal with the multiple-writer paradigm within the B method. Boulmé and Potet [21] proposed an approach inspired by a similar technique of Spec#, where a developer can mark at what places 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 [22] 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. 5(d), because of multiple-writers at the B level and the danger of interferences at the CSP controllers level.

Butler [19] proposed a way of translating CSP systems into action systems, which was later adapted to the B method [20]. 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. Finally, viewing the CSP part of a CSP‖B system as a B entity is possible.

Finally, Event-B—an event-based variant of the classical B—does not provide sharing mechanisms, but some extensions propose sharing solutions to augment the scalability of Event-B: parallel (de-)compoition of events/machines and

their refinement [23], or modularization like in [24]. In addition, let us note that CSP‖Event-B systems have recently been studied wrt. modularity and refinement [25,26]: the deadlock-freeness is ensured under some conditions, and the combined specification refinement guarantees the CSP trace refinement but not the failure refinement.

Perspectives Our proposal allows the relaxation of some constraints upon B machines in a CSP‖B system allowing more flexibility with choosing specification architectures. From there, we conjecture that most architectural patterns can be solved with a combination of our solution and the consistency checking rules of [3]. 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. A version of B extended with rely-guarantee contracts [22] seems to be a good candidate.

Longer-term perspectives include the study of CSP‖B component refinements adapted to our problem. Preliminary studies of recent advances in this domain [27] 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 [3] from PVS to ProB—an animator and model checker for the B method [28], or to a library for the B method in Coq [29], 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: 12th Int. Conf. on Algebraic Methodology and Software Technology (AMAST 2008). Number 5140 in LNCS, Springer-Verlag (2008) 139–144
2. Laleau, R., Semmak, F., Matoussi, A., Petit, D., Hammad, A., Tatibouet, B.: A first attempt to combine sysml requirements diagrams and b. Innovations in Systems and Software Engineering **6** (2010) 47–54
3. Evans, N., Treharne, H.: Interactive tool support for CSP || B consistency checking. Formal Aspects of Computing **19**(3) (2007) 277–302
4. Evans, N., Treharne, H., Laleau, R., Frappier, M.: Applying csp || b to information systems. Software and System Modeling **7**(1) (2008) 85–102
5. Abrial, J.R.: The B Book - Assigning Programs to Meanings. Cambridge University Press (1996)
6. Badeau, F., Amelot, A.: Using B as a high level programming language in an industrial project: Roissy VAL. In: ZB 2005: Formal Specification and Development in Z and B, 4th International Conference of B and Z Users. Volume 3455 of LNCS., Springer-Verlag (2005) 334–354
7. Rouzaud, Y.: Interpreting the B-method in the refinement calculus. In: FM'99: World Congress on Formal Methods. Volume 1709 of LNCS., Springer-Verlag (1999) 411–430
8. Potet, M.L., Rouzaud, Y.: Composition and refinement in the B method. In: B'98 : The 2nd Int. B Conference. (1998) 46–65

9. Roscoe, A.W.: The theory and Practice of Concurrency. Prentice Hall (1997)
10. Schneider, S.A., Treharne, H.E.: CSP theorems for communicating B machines. Formal Aspects of Computing, Special issue of IFM'04 (2005)
11. Evans, N., Treharne, H.E.: Investigating a file transfer protocol using CSP and B. Software and Systems Modelling Journal **4** (2005) 258–276
12. Schneider, S., Cavalcanti, A., Treharne, H., Woodcock, J.: A layered behavioural model of platelets. In: 11th IEEE Int. Conf. on Engieerging of Complex Computer Systems, ICECCS. (2006)
13. Treharne, H., Schneider, S.: Using a process algebra to control B OPERATIONS. In: 1st Int. Conf. on Integrated Formal Methods (IFM'99), York, Springer Verlag (1999) 437–457
14. Schneider, S., Treharne, H.: Communicating B machines. In: Formal specification and development in Z and B (ZB 2002). Volume 2272 of LNCS., Springer Verlag (2002) 416–435
15. 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
16. Colin, S., Lanoix, A., Kouchnarenko, O., Souquières, J.: Using CSP‖B Components: Application to a Platoon of Vehicles. In: 13th Int. ERCIM Wokshop on Formal Methods for Industrial Critical Systems (FMICS 2008). Number 5596 in LNCS, Springer-Verlag (2009) 103–118
17. van Lamsweerde, A.: Goal-driven requirements engineering: the KAOS approach (2009) `http://www.info.ucl.ac.be/~avl/ReqEng.html`.
18. Clearsy: B language reference manual. (2007) v1.8.6.
19. Butler, M.J.: A CSP Approach To Action Systems. PhD thesis, Oxford (1992)
20. Butler, M.: CSP2B : A practical approach to combining CSP and B. In: FM'99: Congress on Formal Methods. Volume 1709 of LNCS., Springer-Verlag (1999) 490–508
21. 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 Int.l B Conf. Volume 4355 of LNCS., Springer (2007) 4–18
22. Büchi, M., Back, R.: Compositional symmetric sharing in B. In: FM'99: Congress on Formal Methods. Volume 1709 of LNCS., Springer-Verlag (1999) 431–451
23. Butler, M.: Decomposition structures for Event-B. In: Integrated Formal Methods, 7th Int. Conf., IFM 2009. Volume 5423 of LNCS., Springer (2009) 20–38
24. Iliasov, A., Troubitsyna, E., Laibinis, L., Romanovsky, A., Varpaaniemi, K., Ilic, D., Latvala, T.: Supporting reuse in Event B development: Modularisation approach. In: Proceedings of ABZ 2010. Volume 5977 of LNCS., Springer (2010)
25. Schneider, S., Treharne, H., Wehrheim, H.: A CSP approach to control in Event-B. In: Integrated Formal Methods. Volume 6396 of Lecture Notes in Computer Science. Springer (2010) 260–274
26. Schneider, S., Treharne, H., Wehrheim, H.: Bounded retransmission in Event-B||CSP: a case study. Electronic Notes in Theoretical Computer Science **280** (2011) 69 – 80 Proceedings of the B 2011 Workshop.
27. Schneider, S., Treharne, H.: Changing system interfaces consistently: A new refinement strategy for CSP||B. Science of Computer Programming **76**(10) (2011) 837 – 860
28. Leuschel, M., Butler, M.: ProB: A Model Checker for B. In: Proc. 12th Int. Conf. FME'2003, Italy. LNCS, Springer-Verlag (2003) 855
29. 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)