Using CSP||B Components: Application to a Platoon of Vehicles

Samuel Colin, Arnaud Lanoix, Olga Kouchnarenko, Jeanine Souquières

Using CSP||B Components: Application to a Platoon of Vehicles [⋆]

Samuel Colin[1], Arnaud Lanoix[1], Olga Kouchnarenko[2], and Jeanine Souquières[1]

[1] LORIA – DEDALE Team – Nancy Université
Campus scientifique
F-54506 Vandoeuvre-Lès-Nancy, France
{firstname.lastname}@loria.fr
[2] LIFC – TFC Team – University of Franche-Comté
16 route de Gray
F-25030 Besançon, France
{firstname.lastname}@lifc.univ-fcomte.fr

Abstract. This paper presents an experience report on the specification and the validation of a real case study in the context of the industrial CRISTAL project. The case study concerns a platoon of a new type of urban vehicles with new functionalities and services. It is specified using the combination, named CSP||B, of two well-known formal methods, and validated using the corresponding support tools. This large – both distributed and embedded – system typically corresponds to a multi-level composition of components that have to cooperate. We identify some lessons learned, showing how to develop and verify the specification and check some properties in a compositional way using theoretical results and support tools to validate this complex system.

Keywords: formal methods, CSP||B, compositional modelling, specification, verification, case study

1 Introduction

This paper is dedicated to an experience report on the specification and the validation of a real case study in the land transportation domain. It takes place in the context of the industrial CRISTAL project which concerns the developpement of a new type of urban vehicles with new functionalities and services. One of its major cornerstones is the development, the validation and the certification of platoon of vehicles. A platoon is a set of autonomous vehicles which have to move in a convoy – i.e. following the path of the leader – through an intangible hooking.

Through the CRISTAL project's collaboration, we have decided to consider each vehicle, named Cristal in the following, as an agent of a Multi-Agent System (MAS). The Cristal driving system perceives information about its environment before producing an instantaneous acceleration passed to its engine. In this context, we consider the platooning problem as a situated MAS which evolves following the Influence/Reaction

[⋆] This work has been partially supported by the French National Research Agency TACOS project, ANR-06-SETI-017 (http://tacos.loria.fr) and by the pôle de compétitivité Alsace/Franche-Comté CRISTAL project (http://www.projet-cristal.net).

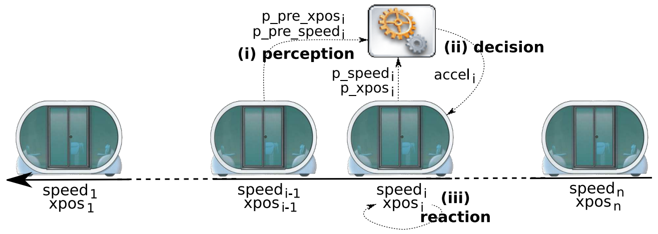

Fig. 1. A platoon of Cristals

model (I/R) [1] in which agents are described separately from the environment. The driving control concerns both a longitudinal control, i.e. maintaining an *ideal* distance between each vehicle, and a lateral control, i.e. each vehicle should follow the track of its predecessor, see Fig. 1. Both controls can be studied independently [2]. At this time, we focus solely on the longitudinal control.

The platoon of Cristal vehicles is a mix of distributed and embedded systems. The former are usually hard to understand and to debug as they can exhibit obscure behaviours. The latter require the satisfaction of safety/security/confidence requirements, alone and when composed together. To address these problems, we reuse the CSP||B framework proposed by Schneider and Treharne [3] of well-established formal methods, B, an environment for the development of provably correct software [4], and CSP (for Communicating Sequential Processes), a process algebra introduced by Hoare [5] for modelling patterns of interactions. We motivate the use of CSP||B by the existence of pure B models describing the agents and vehicles behaviours [6]. By using CSP for coordinating B machines, we aim at giving these B models the architectural, compositional description they lack.

Our approach can be described as a mix between a "bottom-up" and a component-based development. On the one hand, B machines are seen as the smallest abstract components representing various parts of a Cristal vehicle. On the other hand, CSP is used to put these components together, to describe higher-level compounds such as a vehicle or a whole convoy and to make them communicate.

Our first experience with the CSP||B platoon model is presented in a short paper [7]. Here the description of the case study involves detailing two architectural levels. We first consider a single Cristal, then we show how to reuse it to constitute a platoon. Later on we make the model evolve by replacing one component with several others to separate functionalities and refine them[3]. This can be achieved for instance by adapters to connect these new components within the initial architecture [8]. We follow a similar approach, only CSP-oriented. Moreover we use previous theoretical results on CSP||B in an unintended way in this context.

On both the model description and its evolution, we illustrate the relevance of CSP||B for eliminating errors and ambiguities in an assembly and its communication

[3] CSP||B specifications discussed in this paper are available at `http://tacos.loria.fr/platoon-fmics08.zip`

protocols. We are convinced that writing formal specifications can aid in the process of designing autonomous vehicles.

This paper is organised as follows. Section 2 briefly introduces the basic concepts and existing tools on CSP||B. Section 3 presents the specification and the verification process of a single Cristal vehicle whereas Sect. 4 is dedicated to a platoon of vehicles. Section 5 details a vehicle introducing new components, the engine and the location ones. Section 6 presents related works, and Sect. 7 ends with lessons learned from this industrial experience and some perspectives of this development.

2 Basic concepts and tools on CSP||B

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 [4]. The B method has proved its strength in industry with the development of complex real-life applications such as the Roissy VAL [9]. 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 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. Properties are specified in the **INVARIANT** clause of the model, and its evolution is specified by the operations in the **OPERATIONS** clause (see Fig. 3 for an example). The verification of a B model consists in verifying that each operation – assuming its precondition and the invariant hold – satisfies the **INVARIANT**, i.e. the model is *consistent*. Support tools such as B4free (`http://www.b4free.com`) or AtelierB (`http://www.atelierb.eu`) automatically generate proof obligations to ensure the consistency.

A strength of the B method is its stepwise refinement feature: the **REFINEMENT** of a model makes it less indeterministic and more precise with the introduction of more programming language-like features. Refinement can be done until the code of the operations can actually be implemented in a programming language. The consistency of a refinement must also be checked, this time by ensuring that the newly introduced behaviour and/or data do not contradict the model they refine.

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`).

Its denotational semantics is based on the observation of process behaviours. Three kinds of behaviours [10] are observed and well suited for the expression of 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 semantics is associated with a notion of process refinement denoted:

- $\sqsubseteq_T$ for traces refinement;
- $\sqsubseteq_{SF}$ for stable failures refinement and
- $\sqsubseteq_{FD}$ for failures/divergences refinement.

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 [3,11,12]; for case studies, see for example [13,14].

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

$$\boxed{\begin{array}{l} \mathsf{P} ::= \mathsf{c}\ ?\ \mathsf{x}\ !\ \mathsf{v}\ \rightarrow \mathsf{P}\ |\ \ \mathsf{ope}\ !\ \mathsf{v}\ ?\ \mathsf{x} \rightarrow \mathsf{P}\ |\ \ \mathsf{b}\ \&\ \mathsf{P} \\ \qquad\quad |\ \mathsf{P1}\ \square\ \mathsf{P2}\ |\ \ \textbf{if}\ \mathsf{b}\ \textbf{then}\ \mathsf{P1}\ \textbf{else}\ \mathsf{P2}\ |\ \mathsf{S(p)} \end{array}}$$

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.

A controller makes use of *machine channels* which provide the means for controllers to synchronise with the B machine. For each operation $\mathsf{x} \longleftarrow \mathsf{ope(v)}$ of a controlled machine, there is 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.

The behaviour of a guarded process $\mathsf{b}\ \&\ \mathsf{P}$ 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. [3]), the notion of *blocking assertion* is defined by using a guarded process on the inputs of a channel to restrict these inputs: $\mathsf{c}\ ?\ \mathsf{x}\ \&\ \mathsf{E(x)} \rightarrow \mathsf{P}$.

The external choice $\mathsf{P1}\ \square\ \mathsf{P2}$ is initially prepared to behave either as $\mathsf{P1}$ or as $\mathsf{P2}$, with the choice made on the occurrence of the first event. The conditional choice $\textbf{if}\ \mathsf{b}\ \textbf{then}\ \mathsf{P1}\ \textbf{else}\ \mathsf{P2}$ behaves as $\mathsf{P1}$ or $\mathsf{P2}$ depending on b. Finally, $\mathsf{S(p)}$ expresses a recursive call.

Assembling CSP||B components. In addition to the expression of simple processes, CSP provides operators to combine them. The sharing operator $P1 \parallel_E P2$ executes P1 and P2 concurrently, requiring that P1 and P2 synchronise on the events into the sharing alphabet E and allowing independent executions for other events. When combining a CSP controller P and a B machine M associated with P, the sharing alphabet can be dropped ($(P \parallel_{\alpha(M)} M) \equiv P \parallel M$) as there is no ambiguity.

We also consider an indexed form of the sharing operator $\|\|^{i}_{E_i} P(i)$ which executes the processes $P(i)$ in a sharing manner. It is used to build up a collection of similar controlled machines which exchange together.

Verifying CSP||B components. The verification process to ensure the consistency of a controlled machine $(P \parallel M)$ in CSP||B consists in verifying the following conditions:

1. the M machine *consistency* is checked using the B4Free proof tool;
2. the P controller *deadlock-freedom* in the stable-failures model is checked with the FDR2 model-checking tool;
3. the P controller *divergence-freedom* is checked with FDR2;
4. the *divergence-freedom* of $(P \parallel M)$ can be deduced by using a technique based on *Control Loop Invariants* (CLI):
   - P is translated into a B machine $BBODY_P$ using the rewriting rules of [11];
   - a CLI is added to $BBODY_P$;
   - the $BBODY_P$ machine consistency checking is performed with B4Free;
   - by way of [12, Theorem 1], we deduce the *divergence-freedom* of $(P \parallel M)$;
5. by way of [3, Theorem 5.9] and the fact that P is deadlock-free, we deduce the *deadlock-freedom* of $(P \parallel M)$ in the stable failures model.

This verification process can be generalised to achieve the consistency checking of a collection of controlled machines $\|\|^{i}_{E_i} (Pi \parallel Mi)$:

1. we check the divergence-freedom of each $(Pi \parallel Mi)$ as previously;
2. by way of [3, Theorem 8.1], we deduce the divergence-freedom of $\|\|^{i}_{E_i} (Pi \parallel Mi)$;
3. we check the deadlock-freedom of $\|\|^{i}_{E_i} (Pi)$ with FDR2;
4. by way of [3, Theorem 8.6], we deduce the deadlock-freedom of $\|\|^{i}_{E_i} (Pi \parallel Mi)$.

3 Specifying a Single Cristal

As depicted in Fig. 2, in a first approximation, a Cristal vehicle is composed of two parts: the vehicle and its driving system which controls the vehicle. Each part is itself built upon a B machine controlled by an associated CSP process.

3.1 The Vehicle

Specifying the vehicle. The vehicle is a behavioural component reacting to a given acceleration for speeding up or slowing down. It is built upon a Vehicle B machine that describes its inner workings, i.e. its knowledge of speed and location as well as how it updates them w.r.t. a given acceleration, as illustrated in Fig. 3. The $speed \longleftarrow getSpeed()$

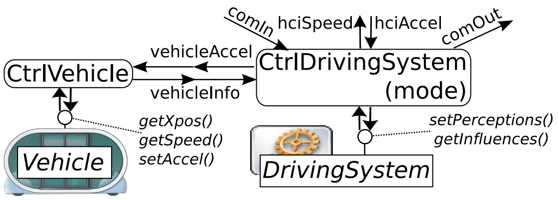

Fig. 2. Architectural view of a Cristal

```
MODEL Vehicle
VARIABLES
  speed, xpos
OPERATIONS
  speed0 ← getSpeed = BEGIN speed0 := speed END ;
  xpos0 ← getXpos = BEGIN xpos0 := xpos END ;
  setAccel(accel) =
   PRE accel ∈ MIN_ACCEL..MAX_ACCEL
   THEN
      ANY new_speed
      WHERE new_speed = speed + accel
      THEN
        IF (new_speed > MAX_SPEED)
        THEN
          xpos := xpos + MAX_SPEED || speed := MAX_SPEED
        ELSE
          IF (new_speed < 0)
          THEN
            xpos := xpos − (speed × speed) / (2 × accel)
            || speed := 0
          ELSE
            xpos := xpos + speed + accel / 2 || speed := new_speed
          END
        END
      END
    END
END
```

Fig. 3. The Vehicle B model

```
REFINEMENT CtrlVehicle_ref
VARIABLES
  xpos_csp, speed_csp, cb
INVARIANT
  xpos_csp ∈ Positions_csp
∧ speed_csp ∈ Speeds_csp
∧ cb = 0
OPERATIONS
CtrlVehicle =
  BEGIN
    CHOICE
      BEGIN
        xpos_csp ← getXpos ;
        speed_csp ← getSpeed ;
        ANY accel_csp WHERE
          accel_csp ∈ Accels_csp
        THEN
          setAccel(accel_csp); cb := 0
        END
      END
    OR
      BEGIN
        speed_csp ← getSpeed ;
        xpos_csp ← getXpos;
        ANY accel_csp WHERE
          accel_csp ∈ Accels_csp
        THEN
          setAccel(accel_csp); cb := 0
        END
      END
    END
  END
END
```

Fig. 4. B rewriting of CtrlVehicle

```
CtrlVehicle =
  ( getXpos ? xpos → getSpeed ? speed → vehicleInfo ! xpos ! speed →
    vehicleAccel ? accel → setAccel ! accel → CtrlVehicle)
  □
  ( getSpeed ? speed → getXpos ? xpos → vehicleInfo ! xpos ! speed →
    vehicleAccel ? accel → setAccel ! accel → CtrlVehicle)
```

Fig. 5. The CtrlVehicle CSP controller

and xpos ← getXpos() methods capture data from the vehicle. The setAccel(accel) method models how the vehicle behaves when passed on a new instantaneous acceleration.

The B machine is made able to communicate by adding a CSP controller, CtrlVehicle, depicted in Fig. 5. It schedules the calls to its various methods. The speed and the location are passed on to the controller through getSpeed ? speed and getXpos ? xpos channels corresponding to invocations of the homonymous methods of the B machine to retrieve the speed and the location of the vehicle. Then, information about speed and location is sent to requesting components through vehicleInfo ! xpos ! speed. Similarly, the controller receives new instantaneous acceleration orders through vehicleAccel ? accel and passes them on through setAccel ! accel to the B machine.

The whole vehicle component with communication facilities is then defined as a parallel composition of the Vehicle machine and its CtrlVehicle controller.

Verifying the vehicle. We follow the verification process given Sect. 2.3 to ensure the consistency of (CtrlVehicle || Vehicle):

- the Vehicle B machine consistency is successfully checked using B4Free;
- the CtrlVehicle controller deadlock-freedom and its divergence-freedom are successfully checked with FDR2;
- Figure 4 illustrates the B rewriting of CtrlVehicle. Its CLI is actually as simple as the $\top$ predicate modulo the typing predicates. This rewriting is shown consistent with B4Free, then (CtrlVehicle || Vehicle) is divergence-free;
- we automatically deduce the deadlock-freedom of (CtrlVehicle || Vehicle).

3.2 The Driving System

Specifying the driving system. The driving system (CtrlDrivingSystem(mode)||Driving System) is built up in a similar way. A DrivingSystem B machine models the decision system: it updates its perceptions and decides for an acceleration passed on to the physical vehicle later on.

Communications are managed by a CtrlDrivingSystem CSP controller shown Fig. 6. It has four running modes corresponding to different uses of a Cristal: SOLO, LEADER of a platoon of Cristals, FOLLOWER of another Cristal into a platoon, and LAST vehicle of a platoon.

In the SOLO mode, the controller requests Cristal speed from the vehicle via vehicle Info ? myXpos ? mySpeed so as to make the HCI displays it (hciSpeed ! mySpeed). It also receives an acceleration from the human driver passed on through hciAccel ? accel and sends this desired acceleration to the vehicle through vehicleAccel ! accel.

The LEADER mode is very similar to the SOLO mode. The only difference consists in additional sending of the Cristal information to the following Cristal via comOut ! mySpeed ! myXpos.

The FOLLOWER mode uses the DrivingSystem B machine: information required by the machine to compute an accurate speed are obtained from the vehicle (vehicleInfo ? myXpos ? mySpeed) and from the leading Cristal (comIn ? preSpeed ? preXpos). Once data are obtained, they are passed on to the B machine through the setPerceptions()

```
DrivingSys_percept(mode) =
   (  (mode == SOLO) &
   vehicleInfo  ? myXpos ? mySpeed → hciSpeed ! mySpeed →  DrivingSys_act(mode) )
   □
   (  (mode == LEADER) &
   vehicleInfo  ? myXpos ? mySpeed → hciSpeed ! mySpeed  → comOut ! mySpeed ! myXpos →
   DrivingSys_act(mode) )
   □
   (  (mode == FOLLOWER) &
   vehicleInfo  ? myXpos ? mySpeed → comIn ? preSpeed ? preXpos → hciSpeed ! mySpeed →
   setPerceptions! myXpos ! mySpeed ! preXpos ! preSpeed → comOut ! mySpeed ! myXpos →
   DrivingSys_act(mode) )
   □
   (  (mode == LAST) &
   vehicleInfo  ? myXpos ? mySpeed → comIn ? preSpeed ? preXpos → hciSpeed ! mySpeed →
   setPerceptions! myXpos ! mySpeed ! preXpos ! preSpeed → DrivingSys_act(mode) )

DrivingSys_act(mode) =
    (  (mode == SOLO) ∨ (mode == LEADER) &
    hciAccel ? accel → vehicleAccel !  accel → DrivingSys_percept(mode) )
    □
   (  (mode == FOLLOWER) ∨ (mode == LAST) &
   getInfluences ? accel → vehicleAccel !  accel → DrivingSys_percept(mode) )

CtrlDrivingSystem(mode) = DrivingSys_percept(mode)
```

Fig. 6. The CtrlDrivingSystem(mode) CSP Controller

method and sent to the following Cristal via comOut ! mySpeed ! myXpos. Otherwise, the acceleration is obtained by a call to the getInfluences() method, and the result is passed on to the vehicle via vehicleAccel ! accel.

The LAST mode is very similar to the FOLLOWER mode. The only difference is that the last vehicle does not send its data to another one.

Verifying the driving system. Using the verification process given Sect. 2.3, the driving system is shown divergence-free and deadlock-free:

- the DrivingSystem B machine is consistent;
- for each mode, the CtrlDrivingSystem(mode) CSP controller is deadlock-free and divergence-free;
- the B rewriting of CtrlDrivingSystem(mode) is consistent.

3.3 The Cristal(mode) Assembly

Specifying the assembly. As illustrated Fig. 2, a Cristal is defined as the parallel composition of a vehicle and its associated driving system, expressed in CSP by:

$$\mathsf{Cristal(mode)} = \big(\mathsf{CtrlVehicle} \parallel \mathsf{Vehicle}\big) \underset{\left\{\left|\substack{\mathsf{vehicleInfo,}\\ \mathsf{vehicleAccel}}\right|\right\}}{\parallel} \big(\mathsf{CtrlDrivingSystem(mode)} \parallel \mathsf{DrivingSystem}\big)$$

Verifying the assembly. Cristal (mode) is shown consistent following the verification process given in Sect. 2.3:

- (CtrlVehicle $\|$ Vehicle) and (CtrlDrivingSystem(mode)$\|$DrivingSystem) are divergence-free, hence Cristal (mode) is also divergence-free;
- (CtrlVehicle $\|$ CtrlDrivingSystem(mode)) is shown deadlock-free with FDR2, then Cristal (mode) is deadlock-free.

Checking a safety property. A safety property we are interested in, states that perception and reaction should alternate while the Cristal runs, i.e. the data are always updated (vehicleInfo) before applying an instantaneous acceleration to the vehicle (vehicleAccel). This property is captured by the following CSP process:

$$\mathsf{Property} = \mathsf{vehicleInfo}\ ?\ \mathsf{xpos}\ ?\ \mathsf{speed} \longrightarrow \mathsf{vehicleAccel}\ ?\ \mathsf{accel} \longrightarrow \mathsf{Property}$$

We need to show that the Cristal meets this property. For that, we first successfully check with FDR2 that there is a trace refinement between the CSP part of Cristal (mode) and Property, i.e. Property $\sqsubseteq_T$ CtrlVehicle $\|$ CtrlDrivingSystem(mode). Then, by applying [3, Corollary 7.2], we obtain that Property $\sqsubseteq_T$ Cristal (mode), i.e. the property is satisfied by the Cristal (mode).

4 Specifying a Platoon of Cristals

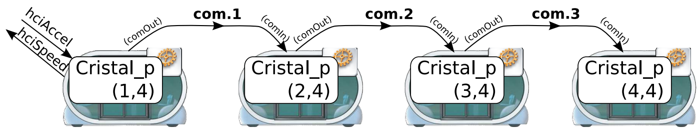

Fig. 7. A Platoon of four Cristals

Once we dispose of a correct model for a single Cristal (mode), we can focus on the specification of a platoon as presented Fig.7. We want the various Cristals to avoid going stale when they move in a platoon. This might happen because a Cristal waits for information from its leading one, i.e. we do not want the communications in the convoy to deadlock.

Specifying the assembly. From the CSP||B specification of a generic Cristal (mode) given in the previous section, we first define a Cristal occupying the position pos into a platoon of max vehicles, as presented Fig. 8: if the Cristal is at the first position, it runs on the LEADER mode, if it is at the last position, it runs on the LAST mode, otherwise, it runs on the FOLLOWER mode. The communication channels are renamed by com.pos/com.pos$-$1, so that the comOut channel of one Cristal matches with the comIn channel of the following Cristal.

```
Cristal_p(pos,max) =
 if (pos == 1)
 then ( Cristal (LEADER) [[ comOut ← com.pos ]] )
 else if (pos == max)
         then ( Cristal (LAST) [[comIn ← com.(pos−1) ]] )
         else ( Cristal (FOLLOWER) [[comIn ← com.(pos−1), comOut ← com.pos ]] )
```

Fig. 8. Cristal_p (pos,max)

A platoon of max Cristals is defined as an assembly of max Cristal_p (pos,max) synchronised on {| com.pos|}, as illustrated Fig. 7 for four vehicles:

$$\text{Platoon(max)} = \underset{\{|\text{com.pos}|\}}{\overset{\text{pos}\in\{1..\text{max}\}}{\Big|\Big|}} \big(\text{Cristal\_p(pos,max)}\big)$$

Verifying the assembly. To check the consistency of Platoon(max), we follow the verification process presented in Sect. 2.3:

- since each Cristal is proved divergence-free, Platoon(max) is divergence-free;
- we have to consider the parallel composition of the CSP parts of all the Cristals. For instance, with four Cristals:

$$\begin{array}{ll}
 & ((\text{CtrlVehicle}||\text{CtrlDrivingSystem(LEADER)})[[\text{comOut} \leftarrow \text{com.1}]]) \\
\underset{\{|\text{com.1}|\}}{||} & ((\text{CtrlVehicle}||\text{CtrlDrivingSystem(FOLLOWER)})[[\text{comIn} \leftarrow \text{com.1, comOut} \leftarrow \text{com.2}]]) \\
\underset{\{|\text{com.2}|\}}{||} & ((\text{CtrlVehicle}||\text{CtrlDrivingSystem(FOLLOWER)})[[\text{comIn} \leftarrow \text{com.2, comOut} \leftarrow \text{com.3}]]) \\
\underset{\{|\text{com.3}|\}}{||} & ((\text{CtrlVehicle}||\text{CtrlDrivingSystem(LAST)})[[\text{comIn} \leftarrow \text{com.3}]])
\end{array}$$

FDR2 checks that this assembly is deadlock-free, hence Platoon(max) is deadlock-free. Consequently, this verification process validates the safety property introduced at the beginning of Sect. 4 saying that the communications, expressed through renaming, should not deadlock.

5 Detailing (CtrlVehicle(mode)||Vehicle)

The definition of the vehicle part presented in Sect. 3.1 is very general. In order to detail information about the vehicle engine and its location, reflecting separation of concerns inside the (CtrlVehicle (mode)||Vehicle) component, we evolve the model presented in Fig. 2 evolve. This evolution introduces new components as illustrated in Fig. 9. They correspond to the following design choices:

1. Now the Vehicle B machine represents the “real” physical vehicle.
2. For compatibility purpose with the rest of the system, the CtrlVehicle is preserved without any modifications.
3. Two new B components are added, modelling two sensors and an actuator, introducing a loss of precision to represent the sensor and actuator effects:

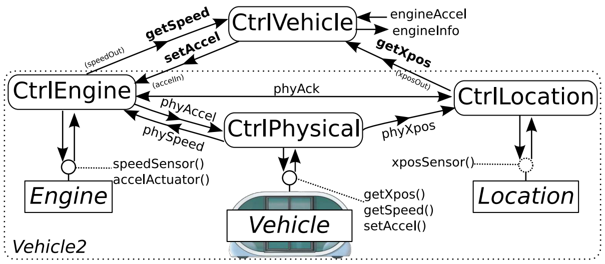

Fig. 9. The Vehicle2 component

```
MODEL Location(er)
OPERATIONS
 p_xpos ← xposSensor(xpos) =
  PRE xpos ∈ ℕ
  THEN
    ANY xx WHERE xx ∈ ℕ
      ∧ xpos − xpos × er / 100 ≤ xx
      ∧ xx ≤ xpos + xpos × er / 100
    THEN
      p_xpos := xx
    END
 END
END
```

Fig. 10. The Location B model

- The B Location machine show Fig. 10 represents an abstract location system able to determine the geographic location of the physical vehicle. It perceives the “real” location and returns an approximated value through p_xpos ← xposSensor(xpos) (with an error of er%). It might be implemented later on by a GPS system, for instance.
- The B Engine machine is introduced to model a speed sensor on the physical vehicle and an acceleration actuator. It senses the “real” speed, returns an approximated value through p_speed ← speedSensor(speed) and applies a decided acceleration order through accel ← accelActuator(d_accel).

4. Three new CSP controllers must be introduced to control the new B machines and to manage communications, i.e. perceptions on the physical world and exchanges between the machines.

5.1 Three New CSP controllers

Specifying CtrlPhysical (Fig. 11). This controller manages the perceptions on the real vehicle. It calls the speed ← getSpeed() and xpos ← getXpos() B methods – to accurate the “real” speed and xpos – and sends these data on phyXpos ! xpos and phySpeed ! speed. It receives a decided acceleration through phyAccel ? accel, then it calls the method setAccel(accel) .

Specifying CtrlLocation (Fig. 12). This controller manages the B Location machine. It perceives the “real” location on phyXpos ? xpos and calls p_xpos ← xposSensor(xpos) to pass them on to the Location component. It sends the *perceived* location through xposOut ! p_xpos.

Specifying CtrlEngine (Fig. 13). This controller is in charge of the Engine B machine, i.e. the speed sensor and the acceleration actuator. A speed perception consists in receiving the “real” speed on phySpeed, passing it on to the B machine by calling the p_speed ← speedSensor(speed) method, and sending the *perceived* speed through

```
CtrlPhysical  =
  (  getSpeed ? speed → phySpeed ! speed → getXpos ? xpos →
     phyXpos ! xpos  → phyAccel ? accel → setAccel ! accel → CtrlPhysical  )
  □
  (  getXpos ? xpos → phyXpos ! xpos → getSpeed ? speed →
     phySpeed ! speed  → phyAccel ? accel → setAccel ! accel → CtrlPhysical )
```

Fig. 11. The CtrlPhysical CSP controller

```
CtrlLocation  =
  phyXpos ? xpos → xposSensor ! xpos ? p_xpos → xposOut ! p_xpos → phyAck → phyAck→ CtrlLocation
```

Fig. 12. The CtrlLocation CSP controller

```
CtrlEngine =
  phySpeed ? speed → speedSensor ! speed ? p_speed → speedOut ! p_speed → phyAck →
  accelIn  ? d_accel → accelActuator ! d_accel ? accel → phyAccel ! accel → phyAck→ CtrlEngine
```

Fig. 13. The CtrlEngine CSP controller

spe edOut ! p_speed. An acceleration setting consists in receiving the decided acceleration on accelIn ? d_accel, passing them on to Engine by calling accel ← accelActuator (d_accel) and sending it to the real vehicle through phyAccel ! accel.

In our first model, speed and location perceptions are done before acceleration is applied. Now, with the separation of concerns introduced by the two components Location and Engine, it would be possible for location perception to be realised *after* an acceleration setting, for instance. In order to ensure this, CtrlEngine and CtrlLocation are synchronised through phyAck.

Verifying the new components. We successfully establish the consistency of (CtrlPhysical || Vehicle), (CtrlEngine || Engine) and (CtrlLocation || Location) using B4Free and FDR2 by following the verification process presented in Sect. 2.3.

5.2 The Vehicle2 Assembly

Vehicle2 is defined as an assembly of the previously detailed components, synchronised on their common channels:

$$\mathsf{Vehicle2} = \left(\begin{array}{c} \left((\mathsf{CtrlEngine} \parallel \mathsf{Engine}) \underset{\{|\mathsf{phyAck}|\}}{\parallel} (\mathsf{CtrlLocation} \parallel \mathsf{Location}) \right) \\ \parallel \\ \left\{ \left| \begin{array}{l} \mathsf{phyAccel}, \\ \mathsf{phySpeed}, \\ \mathsf{phyXpos} \end{array} \right| \right\} \left(\mathsf{CtrlPhysical} \parallel \mathsf{Vehicle} \right) \end{array} \right) \left[\left[\begin{array}{l} \mathsf{accelIn} \leftarrow \mathsf{setAccel}, \\ \mathsf{xposOut} \leftarrow \mathsf{getXpos}, \\ \mathsf{speedOut} \leftarrow \mathsf{getSpeed} \end{array} \right]\right]$$

Some channels have to be renamed to match those of the CtrlVehicle controller.

Verifying that Vehicle2 refines Vehicle. The goal of the Vehicle component evolution is to retain the initial architecture, i.e. we want to replace Vehicle into Cristal (mode) by Vehicle2 and prove that the already established properties are still valid, among which:

- the deadlock-freedom of the whole vehicle (Sect. 3.1);
- the fact that perceptions and actions alternate (Sect. 3.3);
- the deadlock-freedom of the whole convoy (Sect. 4).

Hence Vehicle2 must externally show the same traces as Vehicle and should not introduce new deadlocks. Proving that Vehicle2 refines Vehicle in the stable failures semantics suffices for ensuring that. Indeed, the stable failures refinement preserves safety properties (because it implies trace refinement), liveness properties and deadlock-freedom [10].

We unfortunately face a problem. Vehicle is a B model and Vehicle2 is an assembly of CSP controllers and B machines: there is no manner to check this kind of refinement. To solve this problem, our proposal consists in lifting the refinement checking to an upper level, where refinement is well-defined. In a nutshell, we thus have to prove that the (CtrlVehicle || Vehicle) component is refined by the (CtrlVehicle || Vehicle2) component in the stable failures model which is denoted by:

$$(\mathsf{CtrlVehicle}||\mathsf{Vehicle}) \setminus \alpha(\mathsf{Vehicle}) \sqsubseteq_{SF} (\mathsf{CtrlVehicle}||\mathsf{Vehicle2}) \setminus \alpha(\mathsf{Vehicle})$$

where $\alpha(\mathsf{Vehicle}) \equiv \{|\mathsf{getXpos,getSpeed,setAccel}|\}$.

PROOF:
ASSUME:

$$\mathsf{CtrlVehicle2} = \left(\left(\mathsf{CtrlEngine} \underset{\{|\mathsf{phyAck}|\}}{||} \mathsf{CtrlLocation}\right) \underset{\left\{\left|\begin{array}{l}\mathsf{phyAccel,}\\ \mathsf{phySpeed,}\\ \mathsf{phyXpos}\end{array}\right|\right\}}{||} \mathsf{CtrlPhysical} \right) \left[\left[\begin{array}{l}\mathsf{accelIn} \leftarrow \mathsf{setAccel,}\\ \mathsf{xposOut} \leftarrow \mathsf{getXpos,}\\ \mathsf{speedOut} \leftarrow \mathsf{getSpeed}\end{array}\right]\right]$$

(CtrlVehicle2 is the CSP part of Vehicle2)

1. $(\mathsf{CtrlVehicle} \,||\, \mathsf{Vehicle}) \setminus \alpha(\mathsf{Vehicle}) \sqsubseteq_{SF} (\mathsf{CtrlVehicle} \,||\, \mathsf{CtrlVehicle2}) \setminus \alpha(\mathsf{Vehicle})$

PROOF:

1.1. $\mathsf{CtrlVehicle} \setminus \alpha(\mathsf{Vehicle}) \sqsubseteq_{SF} (\mathsf{CtrlVehicle} \,||\, \mathsf{CtrlVehicle2}) \setminus \alpha(\mathsf{Vehicle})$

(verification carried out by FDR2)

1.2. $(\mathsf{CtrlVehicle} \,||\, \mathsf{Vehicle}) \setminus \alpha(\mathsf{Vehicle}) \sqsubseteq_{SF} \mathsf{CtrlVehicle} \setminus \alpha(\mathsf{Vehicle})$

PROOF:

1.2.1. traces$(((\mathsf{CtrlVehicle} \,||\, \mathsf{Vehicle}) \setminus \alpha(\mathsf{Vehicle})$ = traces$(\mathsf{CtrlVehicle} \setminus \alpha(\mathsf{Vehicle}))$

(definition of traces, hiding of internal channels)

1.2.2. failures$((\mathsf{CtrlVehicle} \,||\, \mathsf{Vehicle}) \setminus \alpha(\mathsf{Vehicle}))$ = failures$(\mathsf{CtrlVehicle} \setminus \alpha(\mathsf{Vehicle})) = \emptyset$

(deadlock-freedom verified by FDR2, [3, theorem 5.9])

1.2.3. $(\mathsf{CtrlVehicle} \,||\, \mathsf{Vehicle}) \setminus \alpha(\mathsf{Vehicle}) \sqsubseteq_{SF} \mathsf{CtrlVehicle} \setminus \alpha(\mathsf{Vehicle})$

(1.2.1, 1.2.2, definition of $\sqsubseteq_{SF}$)

□

1.3. $(\mathsf{CtrlVehicle} \,||\, \mathsf{Vehicle}) \setminus \alpha(\mathsf{Vehicle}) \sqsubseteq_{SF} (\mathsf{CtrlVehicle} \,||\, \mathsf{CtrlVehicle2}) \setminus \alpha(\mathsf{Vehicle})$

(1.1, 1.2, transitivity of $\sqsubseteq_{SF}$)

□

2. $(\mathsf{CtrlVehicle} \,||\, \mathsf{CtrlVehicle2}) \setminus \alpha(\mathsf{Vehicle}) \sqsubseteq_{SF} (\mathsf{CtrlVehicle} \,||\, \mathsf{Vehicle2}) \setminus \alpha(\mathsf{Vehicle})$

PROOF:

2.1. $\mathsf{CtrlVehicle2} \setminus \alpha(\mathsf{Vehicle}) \sqsubseteq_{SF} \mathsf{Vehicle2} \setminus \alpha(\mathsf{Vehicle})$

([3, corollary 8.7] applied to controllers of Vehicle2)

2.2. $(\mathsf{CtrlVehicle} \,||\, \mathsf{CtrlVehicle2}) \setminus \alpha(\mathsf{Vehicle}) \sqsubseteq_{SF} (\mathsf{CtrlVehicle} \,||\, \mathsf{Vehicle2}) \setminus \alpha(\mathsf{Vehicle})$

(2.1, monotonicity of $\sqsubseteq_{SF}$ w.r.t. $||$ and hiding)

□

3. $(\mathsf{CtrlVehicle} \,||\, \mathsf{Vehicle}) \setminus \alpha(\mathsf{Vehicle}) \sqsubseteq_{SF} (\mathsf{CtrlVehicle} \,||\, \mathsf{Vehicle2}) \setminus \alpha(\mathsf{Vehicle})$

(1, 2, transitivity of $\sqsubseteq_{SF}$)

As $(\mathsf{CtrlVehicle} \,||\, \mathsf{Vehicle}) \setminus \alpha(\mathsf{Vehicle}) \sqsubseteq_{SF} (\mathsf{CtrlVehicle} \,||\, \mathsf{Vehicle2}) \setminus \alpha(\mathsf{Vehicle})$ is true, all the properties we wanted to preserve from Vehicle to Vehicle2 are still true: the deadlock-freedom of a vehicle, the deadlock-freedom of the whole convoy as well as the alternation of perceptions and actions. In conclusion, we can replace Vehicle by Vehicle2 without having to check the properties again.

6 Related Works

In addition to works on CSP||B mentioned in Sect. 2, we would like to cite [15], where the authors present a formal framework for verifying distributed embedded systems. An embedded system is described as a set of concurrent real time functions which communicate through a network of interconnected switches involving messages queues and routing services. It presents an abstraction-based verification method which consists in abstracting the communication network by end-to-end timed channels. Proving a given safety property “requires then (1) to prove a set of proof obligations ensuring the correctness of the abstraction step (i.e. the end-to-end channels correctly abstract the network), and (2) to prove ” at the abstract level. The expected advantage of such a method lies on the ability to overcome the combinatorial explosion frequently met when verifying complex systems. This method is illustrated by an avionic case study.

As a comparison point, in [3] Schneider & Treharne illustrate their use of CSP||B with a multi-lift system that can be seen as a distributed system using several instances of a lift, minus the fact that the interactions of the lifts are actually centralised in a dedicated dispatcher. Our goal is very similar, but in contrast to [3], we want to avoid relying on a centralised, or orchestrating, controller.

Similar works exist on structured development with the B method using decomposition, hence in a more “top-down” approach, and refinement. For instance, Bontron & Potet [16] propose a methodology for extracting components out of the enrichments brought by refinement. The extracted components can then be handled to reason about them so as to validate new properties or to detail them more. The interesting point is that their approach stays within the B method framework: this means that the modelling of component communication and its properties has to be done by using the B notation, which can quickly get more cumbersome than an ad-hoc formalism like CSP. Abrial [17] introduces the notion of decomposition of an event system: components are obtained by splitting the specification in the chain of refinements into several specifications expressing different views or concerns about the model. Attiogbé [18] presents an approach dual to the one of Abrial: event systems can be composed with a new asynchronous parallel composition operator, which corresponds to bringing “bottom-up” construction to event systems. In [19], Bellegarde & al. [19] propose a “bottom-up” approach based on synchronisation conditions expressed on the guards of the events. The spirit of the resulting formalism is close to that of CSP||B. Unfortunately, it does not seem to support message passing for communication modelling.

As stated in the introduction, this paper is an evolution of [7]. More precisely, in addition to a more detailed explanation of the specification process we followed with our model, we exploited the renamings of channels so as to give a fitter way for instanciating and assembling several Cristals. We also illustrated a novel use of CSP||B theoretical results: Indeed, theorems about refinement or equivalences of CSP||B components are usually used for easing verification by allowing one to re-express a CSP controller into a simpler one. We used these results to show how to insert new behaviours by splitting up a controller/machine compound without breaking previously verified properties.

7 Conclusion

With the development of a real case study, a platoon of a new type of urban vehicles in the context of the industrial CRISTAL project, we address the importance of formal methods and their utility for highly practical applications. Our contribution mainly concerns methodological aspects for applying known results and tool supports (FDR2 and B4Free). We show how to use the CSP||B framework to compositionally validate the specifications and prove properties of component-based systems, with a precise verification process to ensure the consistency of a controlled machine $(\mathsf{P} \parallel \mathsf{M})$ and its generalisation to a collection of controlled machines $\|^{\mathsf{i}}_{\mathsf{E_i}} (\mathsf{Pi} \parallel \mathsf{Mi})$.

These formal specifications form another contribution of this work. Indeed, having formal CSP||B specifications help – by establishing refinement relations – to prevent incompatibility among various implementations. Moreover, writing formal specifications help in designing a way to manage the multi-level assembly.

This work points out the main drawback of the CSP||B approach: at the interface between the both models, CLIs and augmented B machines corresponding to CSP controllers are not automatically generated. However, this task requires a high expertise level. In our opinion, the user should be able to conduct all the verification steps automatically. Automation of these verification steps could be a direction for future work.

On the case-study side, to go further, we are currently studying new properties such as the non-collision, the non-unhooking and the non-oscillation: which ones are expressible with CSP||B, which ones are tractable and verifiable? This particular perspective is related to a similar work by the authors of CSP||B dealing with another kind of multi-agent system in [14]. So far our use of CSP||B for the platooning model reaches similar conclusions. This nonetheless raises the question of which impact the expression of more complex emerging properties does have on the model.

Further model development requires checking other refinement relations. It also includes evolutions in order to study what happens when a Cristal joins or leaves the platoon, and which communication protocols must be obeyed to do so in a safe manner. We also plan to take into account the lateral control and/or perturbations such as pedestrians or other vehicles.

References

1. Ferber, J., Muller, J.P.: Influences and reaction : a model of situated multiagent systems. In: 2nd Int. Conf. on Multi-agent Systems. (1996) 72–79
2. Daviet, P., Parent, M.: Longitudinal and lateral servoing of vehicles in a platoon. In: Proceeding of the IEEE Intelligent Vehicles Symposium. (1996) 41–46
3. Schneider, S.A., Treharne, H.E.: CSP theorems for communicating B machines. Formal Aspects of Computing, Special issue of IFM'04 (2005)

4. Abrial, J.R.: The B Book. Cambridge University Press (1996)
5. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall (1985)
6. Simonin, O., Lanoix, A., Colin, S., Scheuer, A., Charpillet, F.: Generic Expression in B of the Influence/Reaction Model: Specifying and Verifying Situated Multi-Agent Systems. INRIA Research Report 6304, INRIA (2007)
7. Colin, S., Lanoix, A., Kouchnarenko, O., Souquières, J.: Towards Validating a Platoon of Cristal Vehicles using CSP||B. In: 12th International Conference on Algebraic Methodology and Software Technology (AMAST 2008). LNCS, Springer-Verlag (2008)
8. Lanoix, A., Hatebur, D., Heisel, M., Souquières, J.: Enhancing dependability of component-based systems. In Verlag, S., ed.: Reliable Software Technologies Ada-Europe 2007. Number 4498 in LNCS, Springer Verlag (2007) 41–54
9. 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
10. Roscoe, A.W.: The theory and Practice of Concurrency. Prentice Hall (1997)
11. Treharne, H., Schneider, S.: Using a process algebra to control B OPERATIONS. In: 1st International Conference on Integrated Formal Methods (IFM'99), York, Springer Verlag (1999) 437–457
12. Schneider, S., Treharne, H.: Communicating B machines. In Bert, D., Bowen, J.P., Henson, M.C., Robinson, K., eds.: Formal specification and development in Z and B (ZB 2002). Volume 2272 of LNCS., Springer Verlag (2002) 416–435
13. Evans, N., Treharne, H.E.: Investigating a file transfer protocol using CSP and B. Software and Systems Modelling Journal **4** (2005) 258–276
14. 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)
15. Carcenac, F., Boniol, F.: A formal framework for verifying distributed embedded systems-Frederic based on abstraction methods. Int. J. Softw. Tools Technol. Transf. **8**(6) (2006) 471–484
16. Bontron, P., Potet, M.L.: Automatic construction of validated B components from structured developments. In: Proc. First Int. Conf. ZB'2000, York, Great Britain. Volume 1878 of LNCS., Springer Verlag (2000) 127–147
17. Abrial, J.R.: Discrete system models. Version 1.1 (2002)
18. Attiogbé, C.: Communicating B abstract systems. Research Report RR-IRIN 02.08 (2002) updated july 2003.
19. Bellegarde, F., Julliand, J., Kouchnarenko, O.: Synchronized parallel composition of event systems in B. In Bert, D., Bowen, J.P., Henson, M.C., Robinson, K., eds.: Formal specification and development in Z and B (ZB'2002). Volume 2272 of LNCS., Springer-Verlag (2002) 436–457

A Specifying a Single Cristal

B constants

```
MODEL
   Constants
CONSTANTS
  MAX_SPEED,
  MIN_ACCEL,
  MAX_ACCEL,
  ALERT_DISTANCE,
  IDEAL_DISTANCE
PROPERTIES
  MAX_SPEED ∈ NAT1 ∧
  MIN_ACCEL ∈ INT ∧
  MIN_ACCEL < 0 ∧
  MAX_ACCEL ∈ NAT1 ∧
  ALERT_DISTANCE ∈ NAT ∧
  IDEAL_DISTANCE ∈ NAT ∧
  ALERT_DISTANCE < IDEAL_DISTANCE
ASSERTIONS
  ∀(i , j ).(  (i  ∈ ℤ ∧ j ∈ ℤ ∧ i ≤ j)  ⇒
     (∀k.((k ∈ ℤ) ⇒ (  min({j ,max({i,k })})  ∈ i .. j  ))  )   )
END
```

CSP constants

```
datatype Modes = SOLO | LEADER  | FOLLOWER | LAST
MAX_ID = 10
nametype Ids = {1..MAX_ID}
MAX_SPEED = 0
MIN_ACCEL = 1
MAX_ACCEL = 1
MAX_POS = 0
UNHOOKING_DIST = 1
nametype Speeds = {0..MAX_SPEED}
nametype Accels = {MIN_ACCEL..MAX_ACCEL}
nametype Positions = {0..MAX_POS}

please_compress(p) =
   let
     transparent normalise
   within  normalise(p)
```

CSP channels

```
−− B machine channels between Vehicle∧CtrlVehicle
channel getSpeed∈ Speeds
channel setAccel∈ Accels
channel getXpos∈ Positions

−− B machine channels between DrivingSystem∧CtrlDrivingSystem
channel setPerceptions ∈ Positions . Speeds . Positions .  Speeds
channel getInfluences ∈ Accels

−− Common channels between CtrlVehicle∧CtrlDrivingSystem
channel vehicleInfo∈ Positions .  Speeds
channel vehicleAccel∈ Accels

−− Channels between  an HCI∧CtrlDrivingSystem
channel hciAccel ∈ Accels
channel hciSpeed ∈ Speeds

−− Channels between other cristals∧CtrlDrivingSystem
channel comIn ∈ Speeds . Positions
channel comOut ∈ Speeds . Positions
```

A.1 Specifying (CtrlVehicle || Vehicle)

Vehicle

```
MODEL Vehicle
SEES Constants
VARIABLES
  speed, xpos
INVARIANT
  speed ∈ 0..MAX_SPEED ∧ xpos ∈ ℕ
INITIALISATION
  speed := 0 ∥ xpos :∈ ℕ
OPERATIONS
  speed0 ⟵ getSpeed = BEGIN speed0 := speed END ;
  xpos0 ⟵ getXpos = BEGIN xpos0 := xpos END ;
  setAccel(accel) =
   PRE accel ∈ MIN_ACCEL..MAX_ACCEL
   THEN
      ANY new_speed
      WHERE new_speed = speed + accel
      THEN
        IF (new_speed > MAX_SPEED)
        THEN
          xpos := xpos + MAX_SPEED ∥ speed := MAX_SPEED
        ELSE
          IF (new_speed < 0)
          THEN
            xpos := xpos − (speed × speed) / (2 × accel)
            ∥ speed := 0
          ELSE
            xpos := xpos + speed + accel / 2 ∥ speed := new_speed
          END
        END
      END
    END
END
```

CtrlVehicle

```
CtrlVehicle =
  ( getXpos ? xpos → getSpeed ? speed → vehicleInfo ! xpos ! speed →
    vehicleAccel ? accel → setAccel ! accel → CtrlVehicle )
  □
  ( getSpeed ? speed → getXpos ? xpos → vehicleInfo ! xpos ! speed →
    vehicleAccel ? accel → setAccel ! accel → CtrlVehicle )
```

Abstract B model for CtrlVehicle rewriting

```
MACHINE CtrlVehicle_abs
VARIABLES
  cb
INVARIANT
  cb = 0
INITIALISATION
  cb := 0
OPERATIONS
CtrlVehicle =
  PRE cb = 0
  THEN
    cb := 0
  END
END
```

B CtrlVehicle rewriting

```
REFINEMENT CtrlVehicle_ref
REFINES CtrlVehicle_abs
```

```
INCLUDES Vehicle
SEES
  Constants_csp, Constants
VARIABLES
  xpos_csp, speed_csp, cb
INVARIANT
  xpos_csp ∈ Positions_csp
∧ speed_csp ∈ Speeds_csp
∧ cb = 0
INITIALISATION
   xpos_csp :∈ Positions_csp
|| speed_csp :∈ Speeds_csp
|| cb := 0
OPERATIONS
CtrlVehicle =
  BEGIN
    CHOICE
      BEGIN
        xpos_csp ← getXpos ;
        speed_csp ← getSpeed ;
        ANY accel_csp WHERE
          accel_csp ∈ Accels_csp
        THEN
          setAccel(accel_csp); cb := 0
        END
      END
    OR
      BEGIN
        speed_csp ← getSpeed ;
        xpos_csp ← getXpos;
        ANY accel_csp WHERE
          accel_csp ∈ Accels_csp
        THEN
          setAccel(accel_csp); cb := 0
        END
      END
    END
  END
END
```

A.2 Specifying (CtrlDrivingSystem(mode)||DrivingSystem)

DrivingSystem

```
MODEL DrivingSystem
SEES Constants
VARIABLES
  myXpos, mySpeed,
  preXpos, preSpeed
INVARIANT
  myXpos ∈ ℕ
∧ mySpeed ∈ 0..MAX_SPEED
∧ preXpos ∈ ℕ
∧ preSpeed ∈ 0..MAX_SPEED
INITIALISATION
  myXpos :∈ ℕ
|| mySpeed := 0
|| preXpos :∈ ℕ
|| preSpeed := 0
OPERATIONS
  setPerceptions(myXpos0, mySpeed0, preXpos0, preSpeed0) =
    PRE
      myXpos0 ∈ ℕ
      ∧ mySpeed0 ∈ 0..MAX_SPEED
      ∧ preXpos0 ∈ ℕ
```

```
        ∧ preSpeed0 ∈ 0..MAX_SPEED
      THEN
        myXpos := myXpos0
        ‖ mySpeed := mySpeed0
        ‖ preXpos := preXpos0
        ‖ preSpeed := preSpeed0
      END ;
   accel ← getInfluences =
      IF  (preXpos − myXpos < ALERT_DISTANCE)
      THEN
        accel := MIN_ACCEL
      ELSE
        ANY new_accel
        WHERE
          new_accel = 2 × (preXpos − myXpos) − IDEAL_DISTANCE + preSpeed  − mySpeed
        THEN
          accel :=  min( {MAX_ACCEL, max( {MIN_ACCEL, new_accel} ) } )
        END
      END
END
```

CtrlDrivingSystem

```
DrivingSys_percept(mode) =
   ((mode == SOLO) &
   vehicleInfo  ? myXpos ? mySpeed → hciSpeed ! mySpeed  →  please_compress(DrivingSys_act(mode)) )
   □
   ((mode == LEADER) &
   vehicleInfo  ? myXpos ? mySpeed → hciSpeed ! mySpeed → comOut ! mySpeed ! myXpos →
 please_compress(DrivingSys_act(mode)) )
   □
   ((mode == FOLLOWER) &
   vehicleInfo  ? myXpos ? mySpeed → comIn ? preSpeed ? preXpos → hciSpeed ! mySpeed → setPerceptions! myXpos
!  mySpeed ! preXpos ! preSpeed → comOut ! mySpeed ! myXpos → please_compress(DrivingSys_act(mode)) )
   □
   ((mode == LAST) &
   vehicleInfo  ? myXpos ? mySpeed → comIn ? preSpeed ? preXpos → hciSpeed ! mySpeed → setPerceptions! myXpos
 !  mySpeed ! preXpos ! preSpeed → please_compress(DrivingSys_act(mode)) )
DrivingSys_act(mode) =
    ((mode == SOLO) ∨ (mode == LEADER) &
   hciAccel ? accel → vehicleAccel !  accel → please_compress(DrivingSys_percept(mode)) )
    □
   ((mode == FOLLOWER) ∨ (mode == LAST) &
  getInfluences ? accel → vehicleAccel !  accel → please_compress(DrivingSys_percept(mode)) )
CtrlDrivingSystem(mode) = DrivingSys_percept(mode)
```

Abstract B model for CtrlDrivingSystem rewriting

```
MACHINE CtrlDrivingSystem_abs(Mode)
CONSTRAINTS
   Mode ∈ 1..4
VARIABLES
   cb
INVARIANT
   cb ∈ 0..2
INITIALISATION
   cb := 0
OPERATIONS
DrivingSys_percept =
   PRE cb = 1
   THEN cb :∈ 0..2
   END;
DrivingSys_act =
   PRE cb = 2
   THEN cb :∈ 0..2
   END;
CtrlDrivingSystem =
   PRE cb = 0
   THEN cb :∈ 0..2
   END
END
```

B CtrlDrivingSystem rewriting

```
REFINEMENT CtrlDrivingSystem_ref(Mode)
REFINES CtrlDrivingSystem_abs
SEES Constants_csp, Constants
INCLUDES DrivingSystem
VARIABLES
  accel, cb
INVARIANT
  accel ∈ Accels_csp
∧ cb ∈ 0..2
INITIALISATION
  accel := 0
|| cb := 0
OPERATIONS
DrivingSys_percept =
  BEGIN
    CHOICE
      IF Modes_csp_of_nat(Mode) = SOLO
      THEN
        ANY vehicleInfo_myXpos, vehicleInfo_mySpeed
        WHERE vehicleInfo_myXpos ∈ Positions_csp
            ∧ vehicleInfo_mySpeed ∈ Speeds_csp
        THEN cb := 1
        END
      ELSE SELECT TRUE = FALSE THEN skip END
      END
    OR
      IF Modes_csp_of_nat(Mode) = LEADER
      THEN
        ANY vehicleInfo_myXpos, vehicleInfo_mySpeed
        WHERE vehicleInfo_myXpos ∈ Positions_csp
            ∧ vehicleInfo_mySpeed ∈ Speeds_csp
        THEN cb := 1
        END
      ELSE SELECT TRUE = FALSE THEN skip END
      END
    OR
      IF Modes_csp_of_nat(Mode) = FOLLOWER
      THEN
        ANY vehicleInfo_myXpos, vehicleInfo_mySpeed
        WHERE vehicleInfo_myXpos ∈ Positions_csp
            ∧ vehicleInfo_mySpeed ∈ Speeds_csp
        THEN
          ANY comIn_preSpeed, comIn_preXpos
          WHERE comIn_preSpeed ∈ Speeds_csp
              ∧ comIn_preXpos ∈ Positions_csp
          THEN
            setPerceptions(vehicleInfo_myXpos, vehicleInfo_mySpeed,
                           comIn_preXpos, comIn_preSpeed);
            cb := 1
          END
        END
      ELSE SELECT TRUE = FALSE THEN skip END
      END
    OR
      IF Modes_csp_of_nat(Mode) = LAST
      THEN
        ANY vehicleInfo_myXpos, vehicleInfo_mySpeed
        WHERE vehicleInfo_myXpos ∈ Positions_csp
            ∧ vehicleInfo_mySpeed ∈ Speeds_csp
        THEN
          ANY comIn_preSpeed, comIn_preXpos
          WHERE comIn_preSpeed ∈ Speeds_csp
              ∧ comIn_preXpos ∈ Positions_csp
          THEN
            setPerceptions(vehicleInfo_myXpos, vehicleInfo_mySpeed,
                           comIn_preXpos, comIn_preSpeed);
            cb := 1
          END
        END
      ELSE SELECT TRUE = FALSE THEN skip END
      END
    END
  END;
DrivingSys_act =
  BEGIN
    CHOICE
      IF Modes_csp_of_nat(Mode) = SOLO ∨ Modes_csp_of_nat(Mode) = LEADER
      THEN
        ANY hciAccel_accel
        WHERE hciAccel_accel ∈ Accels_csp
        THEN cb := 0
        END
      ELSE SELECT TRUE = FALSE THEN skip END
      END
    OR
      IF Modes_csp_of_nat(Mode) = FOLLOWER ∨ Modes_csp_of_nat(Mode) = LAST
      THEN
        accel ← getInfluences;
        cb := 0
      ELSE SELECT TRUE = FALSE THEN skip END
      END
    END
  END;
CtrlDrivingSystem =
  BEGIN
    cb := 0
  END
END
```

A.3 Specifying the assembly Cristal(mode)

CSP part of Cristal (mode)

```
Cristal (mode) =
( CtrlVehicle [| {| vehicleInfo , vehicleAccel |} |] CtrlDrivingSystem(mode) )
```

Safety property

```
Property = vehicleInfo ? xpos ? speed → vehicleAccel ? accel → Property
```

B Specifying a Platoon of Cristals

CSP part of Cristal_p (pos,max)

```
channel com ∈ Ids . Speeds . Positions

Cristal_x (mode) = Cristal (mode)
    \ {| vehicleInfo , vehicleAccel, setPerceptions, getInfluences,
         getSpeed, getXpos,setAccel, hciSpeed, hciAccel |}

Cristal_p (pos,max) =
  if (pos == 1)
  then (  Cristal_x(LEADER) [[ comOut ← com.1 ]] )
  else if  (pos == max)
         then (  Cristal_x(LAST) [[comIn ← com.(max−1) ]] )
         else (  Cristal_x(FOLLOWER) [[comIn ← com.(pos−1), comOut ← com.pos ]] )
```

CSP part of Platoon(max)

```
Platoon(max) =  (  ||   pos∈{1..max} @ [ {|com.pos|} ] Cristal_p(pos,max) )
```

C Detailing (CtrlVehicle(mode)||Vehicle)

CSP channels

```
−− channels between CtrlVehicle∧CtrlEngine/CtrlLocation
channel accelIn ∈ Accels
channel speedOut ∈ Speeds
channel xposOut ∈ Positions

−− B machine channels between Engine∧CtrlEngine
channel speedSensor ∈ Speeds . Speeds
channel accelActuator ∈ Accels . Accels

−− B machine channel between Location∧CtrlLocation
channel xposSensor ∈ Positions . Positions

−− channels between CtrlEngine∧CtrlPhysical
channel phyAccel ∈ Accels
channel phySpeed ∈ Speeds

−− channel between CtrlLocation∧CtrlPhysical
channel phyXpos ∈ Positions

−− channel between CtrlEngine∧CtrlLocation
channel phyAck
```

C.1 Specifying (CtrlPhysical||Vehicle)

CtrlPhysical

```
CtrlPhysical  =
  (  getSpeed ? speed → phySpeed ! speed → getXpos ? xpos →
     phyXpos ! xpos  → phyAccel ? accel → setAccel ! accel → CtrlPhysical  )
  □
  (  getXpos ? xpos → phyXpos ! xpos → getSpeed ? speed →
     phySpeed ! speed  → phyAccel ? accel → setAccel ! accel → CtrlPhysical )
```

Abstract B model for CtrlPhysical rewriting

```
MACHINE CtrlPhysical_abs
VARIABLES
  cb
INVARIANT
  cb ∈ 0..0
INITIALISATION
  cb := 0
OPERATIONS
CtrlPhysical  =
  PRE cb = 0
  THEN cb :∈ 0..0
  END
END
```

B CtrlPhysical rewriting

```
REFINEMENT CtrlPhysical_ref
REFINES CtrlPhysical_abs
SEES Constants_csp, Constants
INCLUDES Vehicle
VARIABLES
  p_xpos,p_speed, cb
INVARIANT
  p_xpos ∈ Positions_csp
∧ p_speed ∈ Speeds_csp
∧ cb ∈ 0..0
INITIALISATION
   p_xpos := 0
‖ p_speed := 0
‖ cb := 0
OPERATIONS
CtrlPhysical =
  BEGIN
    CHOICE
      BEGIN
        p_speed ← getSpeed;
        p_xpos ← getXpos;
        ANY accel
        WHERE accel ∈ Accels_csp
        THEN
          setAccel(accel);
          cb:=0
        END
      END
    OR
      BEGIN
        p_xpos ← getXpos;
        p_speed ← getSpeed;
        ANY accel
        WHERE accel ∈ Accels_csp
        THEN
          setAccel(accel);
          cb:=0
        END
      END
    END
  END
END
```

C.2 Specifying (CtrlEngine‖Engine)

Engine

```
MODEL Engine(ac_er,sp_er)
CONSTRAINTS ac_er ∈ 0..100 ∧ sp_er ∈ 0..100

SEES Constants

OPERATIONS

  p_speed ← speedSensor(speed) =
    PRE speed ∈ 0..MAX_SPEED
    THEN
      ANY sp WHERE sp ∈ 0..MAX_SPEED
      ∧ speed − speed × sp_er / 100 ≤ sp
      ∧ sp ≤ speed + speed × sp_er / 100
      THEN
        p_speed := sp
      END
    END;

  accel ← accelActuator(d_accel) =
    PRE d_accel ∈ MIN_ACCEL..MAX_ACCEL
    THEN
      ANY ac WHERE ac ∈ MIN_ACCEL..MAX_ACCEL
      ∧ d_accel − d_accel × ac_er / 100 ≤ ac
      ∧ ac ≤ d_accel + d_accel × ac_er / 100
      THEN
        accel := ac
      END
    END

END
```

CtrlEngine

```
CtrlEngine =
  phySpeed ? speed → speedSensor ! speed ? p_speed → speedOut ! p_speed → phyAck →
  accelIn ? d_accel → accelActuator ! d_accel ? accel → phyAccel ! accel → phyAck→ CtrlEngine
```

Abstract B model for CtrlEngine rewriting

```
MACHINE CtrlEngine_abs
VARIABLES
  cb
INVARIANT
  cb ∈ 0..0
INITIALISATION
  cb := 0
OPERATIONS
CtrlEngine =
  PRE cb = 0
  THEN cb :∈ 0..0
  END
END
```

B CtrlEngine rewriting

```
REFINEMENT CtrlEngine_ref
REFINES CtrlEngine_abs
SEES Constants_csp, Constants
INCLUDES Engine(1,2)
VARIABLES
  speed0, accel0, cb
INVARIANT
  speed0 ∈ Speeds_csp
∧ accel0 ∈ Accels_csp
∧ cb ∈ 0..0
INITIALISATION
  speed0 := 0
∥ accel0 := 0
∥ cb := 0
OPERATIONS
CtrlEngine =
  BEGIN
    ANY speed
    WHERE speed ∈ Speeds_csp
    THEN
      speed0 ← speedSensor(speed);
      ANY accel
      WHERE accel ∈ Accels_csp
      THEN
        accel0 ← accelActuator(accel);
        cb := 0
      END
    END
  END
END
```

C.3 Specifying (CtrlLocation||Location)

Location

```
MODEL Location(er)
CONSTRAINTS er ∈ 0..100
SEES Constants
OPERATIONS
 p_xpos ← xposSensor(xpos) =
  PRE xpos ∈ ℕ
  THEN
    ANY xx WHERE xx ∈ ℕ
     ∧ xpos − xpos × er / 100 ≤ xx
     ∧ xx ≤ xpos + xpos × er / 100
    THEN
     p_xpos := xx
    END
 END
END
```

CtrlLocation

```
CtrlLocation =
  phyXpos ? xpos → xposSensor ! xpos ? p_xpos → xposOut ! p_xpos → phyAck → phyAck→ CtrlLocation
```

Abstract B model for CtrlLocation rewriting

```
MACHINE CtrlLocation_abs
VARIABLES
  cb
INVARIANT
  cb ∈ 0..0
INITIALISATION
  cb := 0
OPERATIONS
CtrlLocation =
  PRE cb = 0
  THEN cb :∈ 0..0
  END
END
```

B CtrlLocation rewriting

```
REFINEMENT CtrlLocation_ref
REFINES CtrlLocation_abs
SEES Constants_csp, Constants
INCLUDES Location(3)
VARIABLES
  xpos0, cb
INVARIANT
  xpos0 ∈ Positions_csp
∧ cb ∈ 0..0
INITIALISATION
  xpos0 := 0
|| cb := 0
OPERATIONS
CtrlLocation =
  BEGIN
    ANY xpos
    WHERE xpos ∈ Positions_csp
    THEN
```

```
      xpos0 ⟵ xposSensor(xpos);
      cb := 0
    END
  END
END
```

C.4 Specifying the assembly Vehicle2

CSP part of Vehicle2

```
CtrlVehicle2 = ( ( CtrlEngine [| {| phyAck |} |] CtrlLocation )
 [| {| phyAccel, phySpeed, phyXpos |} |] CtrlPhysical )
        \ {| phySpeed, phyAccel, phyXpos, phyAck,
              speedSensor, accelActuator, xposSensor,
              getSpeed, getXpos, setAccel |}

CtrlVehicle2_x = please_compress(
      CtrlVehicle2 [[ accelIn ⟵ setAccel, speedOut ⟵ getSpeed, xposOut ⟵ getXpos ]]
)

Vehicle2 = CtrlVehicle [| {| setAccel, getSpeed, getXpos |} |] CtrlVehicle2_x
```