Electronic Communications of the EASST
Volume 23 (2009)

Proceedings of the Ninth International Workshop on Automated Verification of Critical Systems (AVOCS 2009)

Mobile CSP||B

Beeta Vajar, Steve Schneider and Helen Treharne

17 pages

Guest Editor: Markus Roggenbach
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Mobile CSP||B

Beeta Vajar[1], Steve Schneider[2] and Helen Treharne[3]

[1]b.vajar@surrey.ac.uk
[2]s.schneider@surrey.ac.uk
[3]h.treharne@surrey.ac.uk
Department of Computing, University of Surrey, Guildford, Surrey, UK

Abstract: CSP||B is a combination of CSP and B in which CSP processes are used as control executives for B machines. This architecture enables a B machine and its controller to interact and communicate with each other while working in parallel. The architecture has focused on sequential CSP processes as dedicated controllers for B machines. This paper introduces Mobile CSP||B, a formal framework based on CSP||B which enables us to specify and verify concurrent systems with mobile architecture instead of the previous static architecture. In Mobile CSP||B, a parallel combination of CSP processes act as the control executive for the B machines and these B machines can be transferred between CSP processes during the system execution. The paper introduces the foundations of the approach, and illustrates the result with an example.

Keywords: CSP, B, mobility

1 Introduction

Numerous methods which combine state and event based models have been proposed: ZCCS [GS97], CSP-OZ [Fis97], Circus [OCW07], CSP2B [But00], ProB [LB03] and *CSP* || *B* [TS02, ST05]. Their advantage is that complex systems can be described and their verification ensures consistency of the models. Some integration, e.g., PIOZ [TDC04] and $\pi \mid B$ [KST07], support the description of mobility and dynamic patterns. This additional functionality is suitable for modelling agent systems or peer-to-peer networks where consideration of mobility is important. In this paper, we are interested in extending our *CSP* || *B* approach to include mobility and we have developed a formal framework so that we can compositionally verify the consistency of *CSP* || *B* specifications that include mobile aspects. This allows us to extend the range of specifications that we can write in *CSP* || *B* and retains our philosophy of not changing the underlying CSP [Sch00, Hoa85] and classical B [Sch01, Abr96] semantics. The framework adopts similar concepts to the architecture proposed in $\pi \mid B$. However, $\pi \mid B$ was limited to systems without inputs and outputs and was restricted to a framework to support divergence freedom verification. In this paper, we can deal with specifications that contain inputs and outputs and in addition to divergence freedom we can check for deadlock freedom. These two checks are the minimum verification that should be carried out in order to ensure that a mobile *CSP* || *B* specification is consistent. We use the divergence freedom check to confirm that B operations are called within their preconditions. In [Ros08] Roscoe introduces a new operator into a variant of CSP, introducing mobility in the way that the rights to use particular events are transferred between processes

along special rights channels. Our approach is similar, in that channels can be transferred between processes. However, our work is motivated by the desire to retain access to the supporting CSP and B toolsets, and so we aim to minimise the extension required to enable the form of mobility we aim to model.

1.1 The B-Method

The main unit of specification in the B-Method is an *abstract machine*. An abstract machine describes the state of the system in terms of mathematical structures such as sets, relations, functions and sequences. It also provides operations which change the state of the system. Each operation has a precondition or a guard. For the purposes of this paper we will restrict ourselves to preconditioned operations, as in classical B. Preconditioned operations have the form **PRE** P **THEN** S **END**, where P is the precondition, and S is the body of the operation, written in *Abstract Machine Notation (AMN)*, a simple language that contains assignment, choice, conditional, and precondition statements. A precondition expresses a predicate on the state of the machine which must hold when the operation is invoked, in order to ensure that the operation behaves as described by S; otherwise no guarantees can be given.

If S is an statement and Q is a predicate, the notation $[S]Q$ (also $wp(S,Q)$) denotes the weakest precondition which must be true when executing S to guarantee to reach a state in which Q is true.

The B-Method is a formal method supported by many comprehensive tools such as: B-Toolkit [BC02], ProB, and Atelier B [Cle09].

1.2 CSP

CSP is a theoretical notation or language for specifying and verifying concurrent systems, in terms of the events that they can perform. CSP provides a framework for describing and analysing interacting aspects of concurrent systems. Concurrent systems consist of interacting components known as processes. Each process works independently and may interact with its environment and other processes in the system. A process performs various events which describe its behaviour. CSP is an event-based formal language for designing and analysing a system behaviour through the events happening in the system. Its operators include event prefixing, channel input and output, choice, recursion, and parallel composition in which parallel components synchronise on events that they have in common. The variant of CSP that we will use in this paper is given in Section 3.

CSP has a variety of semantic models, based on observations. In this paper we are concerned primarily with traces and with divergences, though also with a need to handle deadlocks (which requires the failures model). A *trace* tr of a process P is a finite sequence of events which P is able to perform. The set of all possible traces of process P is denoted by $traces(P)$. A *divergence* of a process P is a sequence of events tr during or after which P can diverge—no guarantees can be made of its behaviour after divergence.

CSP is supported by highly efficient software tools such as ProBE [FSEL07b] and FDR [FSEL07a], supporting state exploration, refinement, divergence, and deadlock checking.

1.3 CSP||B

CSP || *B* is a parallel combination between CSP and B in which a CSP process is used as a control executive for a B machine. For each B machine's operation $bb \longleftarrow op(aa)$, there is a channel *op* between the CSP controller and the B machine which carries data types the same as the types of *aa* and *bb*. This provides the means for CSP controller and its controlled B machine to synchronise and communicate with each other while working in parallel. A B machine and its controller can send or receive values from each other through these channels. For instance, the CSP controller sends the value of *aa* to the B machine and receives the B machine's output, *bb*, through channel *op*. This means that in addition to control the execution order of the B operations, CSP controller and B machine can communicate with each other and they can exchange data and information through these channels while working in parallel in the system.

In [Mor90], Morgan introduces traces, failures and divergences semantics of CSP for action systems by using weakest precondition formulae. Based on this achievement, CSP semantics of traces, failures and divergences have been defined for B machines in [ST05]. Thus, a B machine can be understood as a CSP process. This common semantic framework makes it possible to define the parallel combination of B machines and CSP processes. In this framework the invocation of an operation outside its precondition corresponds to divergence.

According to the definitions in [Mor90], a trace of a B machine is a finite sequence of its operations. Divergence happens in a B machine when a pre-conditioned operation is called outside its precondition.

1.4 Introducing Mobility

In *CSP* || *B*, each CSP process can be the control executive of only one B machine and each B machine has only one CSP process as its controller. The architecture has focused on sequential CSP processes as dedicated controllers for B machines. The objective of this paper is to generalise *CSP* || *B* architecture in designing a new framework, *Mobile CSP* || *B*, which enables us to describe and verify systems in which a parallel combination of CSP processes are collectively the controllers of B machines, and each single B machine can be controlled by different CSP processes during the execution. By introducing mobility, each CSP process can receive a (mobile) machine or give it to another CSP process during the execution. An example of these kinds of systems is peer-to-peer networks in which data (B machines) can be transfered between the connected nodes (CSP controllers).

The following step is the consistency verification. We must ensure that B operations are always called within their preconditions, as they are passed between the controllers. We provide a theorem to establish divergence freedom of the whole mobile combined communicating system containing several CSP controllers each controlling several B machines, by establishing properties for each CSP controller separately. We also have the result that deadlock-freedom of the controllers implies deadlock-freedom of the combined system.

2 Mobile CSP || B

In standard CSP||B, a controlled component consists of a CSP controller P in parallel with a B machine M. Operations op with inputs s and outputs t are declared in machines M as $t \longleftarrow op(s)$. In the combination they are treated as channels $op.s.t$. Standard CSP||B has a static architecture in which one B machine works in parallel with only one CSP controller and each CSP controller can be the controller of only one B machine. So, the behaviour of the parallel combination is predictable as we have fixed controlled components during the system execution.

In Mobile CSP||B, we intend to create a mobile architecture in which B machines are able to be transferred from one controller to another controller and each controller can work with more than one B machine at the same time. As controllers can exchange B machines between each other, B machines can have different controllers during their execution.

To enable machines to be passed around the system, we introduce a unique machine channel called *machine references*. CSP controllers use machine references as the link to interact with B machines. A machine reference is the only channel through which a CSP controller and a B machine can communicate with each other. As a result, a controller is only able to work with a machine if it owns that machine's reference. In other words, possession of a machine means having that machine's reference. In order for machines to be exchanged between controllers, machine references must be passed around between controllers in the system. Therefore, when a machine is going to be passed from one controller to another, the sender controller passes that B machine's reference to the other controller, as illustrated on the right in Figure 1. It shows that B machine M_1 is passed from CSP controller P_1 to P_2. The figure also shows the difference between Static CSP||B architecture and Mobile CSP||B architecture.

B machine M with machine reference z is presented in the system as $z : M$. All operations op in M are replaced with $z.op$. So, operation calls of the machine $z : M$ correspond to the communication $z.op.s.t$, and the machine reference z can itself be passed between controllers.

We introduce channels called *control points* between pairs of controllers on which machine references are passed around. When a machine is passed from one controller to another, the sender controller passes that B machine's reference to the other controller through the control point channel which exists between those two controllers.

We require that only one CSP controller is in possession of z at any one time, so that when z is passed from P_1 to P_2 then P_1 is no longer able to use z to call the operations of the machine. This will be the cornerstone for reasoning about the action of controllers on a mobile machine: that a controller has exclusive control over a machine it is using, and other controllers cannot interfere with its use of the machine.

We introduce MR as the set of machine references, CP as the set of control points, and C as the set of regular CSP channels. Each channel c in the set of regular channels C has a *type* denoted $type(c)$. The type of channels in CP is MR. Each machine reference in MR is associated with a particular B machine. The type of a machine reference z is the set of operations (with inputs and outputs) of the unique machine M that is associated with z.

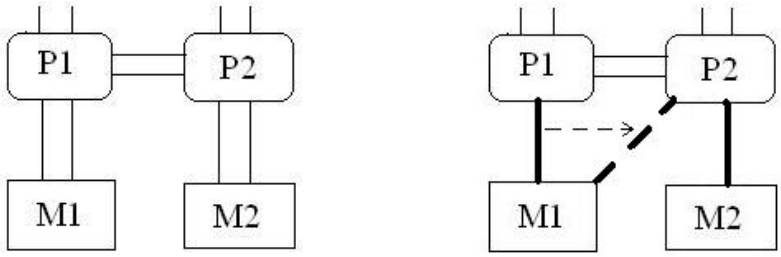

Figure 1: Static CSP||B architecture and Mobile CSP||B architecture

3 Mobile CSP Controllers

We will use the name *LOOP* to denote a mobile CSP controller. A process *LOOP* has a set of static *channels* $\chi(LOOP)$, which contains its communication channels and control points. Any particular control point in the alphabet of *LOOP* will be either incoming or outgoing with respect to *LOOP*, and is not permitted to be both. We identify the incoming control points within $\chi(LOOP)$ as $\chi_i(LOOP)$. The outgoing control points within $\chi(LOOP)$ are denoted $\chi_o(LOOP)$. The alphabet associated with communication channels is denoted $\chi_c(LOOP)$. These three sets are pairwise disjoint, and their union is $\chi(LOOP)$.

The syntax of mobile CSP controllers is defined by the following BNF:

$$
\begin{array}{lll}
P \quad ::= & SKIP \mid c?x \rightarrow P(x) \mid c!v \rightarrow P & \text{termination; communication} \\
& \mid cp_1?w \rightarrow P(w) \mid cp_2!z \rightarrow P \;\; (z \notin fv(P)) & \text{passing machine references} \\
& \mid z.op!s?t \rightarrow P(t) & \text{operation call} \\
& \mid P' \,\Box\, P'' \mid P' \sqcap P'' \mid \mathit{if}\ b\ \mathit{then}\ P'\ \mathit{else}\ P'' & \text{choice} \\
& \mid e \rightarrow P & \text{prefix} \\
& \mid N(E_1,\ldots,E_n) & \text{recursive call}
\end{array}
$$

where b is a boolean expression, e is an atomic CSP event which is not a B operation, $c \in \chi_c(LOOP)$, $cp_1 \in \chi_i(LOOP)$, $cp_2 \in \chi_o(LOOP)$, v is a variable of type $type(c)$, z and w are variables of type *MR*, $t \longleftarrow op(s)$ is an operation of the B machine associated with z. $fv(P)$ is the set of free variables in P, including variables for machine references. In $N(E_1,\ldots,E_n)$, each expression E_i either does not mention *MR* variables at all, or else is an *MR* variable; and no *MR* variable in the list is repeated.

Sequential processes are then defined recursively as follows:

$$N_i(w_{i1},\ldots,w_{in}) \mathrel{\widehat{=}} P_i \qquad \text{where } fv(P_i) \subseteq \{w_{i1},\ldots,w_{in}\}$$

LOOP is then defined as some $N_i(m_{i1},\ldots,m_{in})$ for values $m_{i1},\ldots,m_{in}$, where all instantiations of *MR* variables are distinct.

The machine references that N_i knows initially will appear in the list $m_{i1},\ldots,m_{in}$. The set of machine references, mr, owned by $LOOP$ can be defined by $mr(LOOP) = \{m_{i1},\ldots,m_{in}\} \cap MR$.

4 Parallel combination

A mobile combined communicating system including n controllers and m B machines is represented as

$$LOOP_1 \,||\, LOOP_2 \,||\, \ldots \,||\, LOOP_n \,||\, z_1 : M_1 \,||\, z_2 : M_2 \,||\, \ldots \,||\, z_m : M_m$$

where $z_1, z_2, \ldots z_m$ are the machine references for B machines $M_1, M_2, \ldots M_m$ respectively, and $i \neq j \Rightarrow z_i \neq z_j$

Mutual recursive CSP processes can be composed in parallel, only if (1) they have no machine references in common: $\forall 1 \leqslant i,j \leqslant n \bullet mr(LOOP_i) \cap mr(LOOP_j) = \emptyset$, (2) they differ on their incoming control points and their outgoing control points: $\forall 1 \leqslant j,k \leqslant n \bullet \chi_i(LOOP_j) \cap \chi_i(LOOP_k) = \emptyset$, $\forall 1 \leqslant j,k \leqslant n \bullet \chi_o(LOOP_j) \cap \chi_o(LOOP_k) = \emptyset$, and (3) each control point in the system has both a sender and a receiver. In other words, any outgoing (or incoming) control point in one controller is an incoming (or outgoing) control point of one of the other controllers in the system: $\bigcup_{j=1}^{n} \chi_i(LOOP_j) = \bigcup_{i=1}^{n} \chi_o(LOOP_i)$.

The free variables of the parallel combination of controllers is given as follows:

$$fv(LOOP_1 \,||\, LOOP_2 \,||\, \ldots \,||\, LOOP_n) = \bigcup_{i=1}^{n} fv(LOOP_i)$$

When a system is constructed, each machine reference must be given a different concrete value.

The alphabets for the parallel combination of controllers are given as follows:

$$\chi_i(LOOP_1 \,||\, LOOP_2 \,||\, \ldots \,||\, LOOP_n) = \bigcup_{j=1}^{n} \chi_i(LOOP_j)$$

$$\chi_o(LOOP_1 \,||\, LOOP_2 \,||\, \ldots \,||\, LOOP_n) = \bigcup_{i=1}^{n} \chi_o(LOOP_i)$$

$$\chi_c(LOOP_1 \,||\, LOOP_2 \,||\, \ldots \,||\, LOOP_n) = \bigcup_{i=1}^{n} \chi_c(LOOP_i)$$

The language of process terms and the rules for parallel combination of controllers have been designed to ensure that at any point in the system execution, only one controller has possession of any machine reference. Controllers do not share any machine references to begin with, and when a machine reference is passed along a control point to another controller, it is not retained by the sending controller.

In order to define the traces of parallel composition, it is necessary to keep track of the machine references as they are used and passed between controllers. We can define the projection of a trace onto a particular controller $LOOP$ given the channels $\chi_i(LOOP)$, $\chi_o(LOOP)$, $\chi_c(LOOP)$, provided we also know the set of machine references mr owned by the controller. This definition is based on the corresponding definition from [VST07].

The projection of a trace tr onto $\chi(LOOP)$ and a set of machine references mr can be defined inductively as shown in Figure 2 where $tr \upharpoonright \chi(LOOP), mr$ means the projection of tr onto alphabet of controller $LOOP$ who owns the set of machine references mr. This enables a definition of

$$
\begin{array}{rcl}
\langle\rangle \upharpoonright \chi(LOOP), mr & = & \langle\rangle \\
(\langle cp.z\rangle \frown tr) \upharpoonright \chi(LOOP), mr & = & \begin{cases} \langle cp.z\rangle \frown (tr \upharpoonright \chi(LOOP), mr \cup \{z\}) & \text{if } cp \in \chi_i(LOOP) \wedge z \notin mr \\ \langle cp.z\rangle \frown (tr \upharpoonright \chi(LOOP), mr - \{z\}) & \text{if } cp \in \chi_o(LOOP) \wedge z \in mr \\ tr \upharpoonright \chi(LOOP), mr & \text{if } cp \notin \chi_i(LOOP) \cup \chi_o(LOOP) \wedge z \notin mr \\ \text{undefined} & \text{otherwise} \end{cases} \\
(\langle c\rangle \frown tr) \upharpoonright \chi(LOOP), mr & = & \begin{cases} \langle c\rangle \frown (tr \upharpoonright \chi(LOOP), mr) & \text{if } c \in \chi_c(LOOP) \\ tr \upharpoonright \chi(LOOP), mr & \text{if } c \notin \chi_c(LOOP) \end{cases} \\
(\langle z.op\rangle \frown tr) \upharpoonright \chi(LOOP), mr & = & \begin{cases} \langle z.op\rangle \frown (tr \upharpoonright \chi(LOOP), mr) & \text{if } z \in mr \\ tr \upharpoonright \chi(LOOP), mr & \text{if } z \notin mr \end{cases}
\end{array}
$$

Figure 2: Projection of a trace onto $\chi(LOOP), mr$

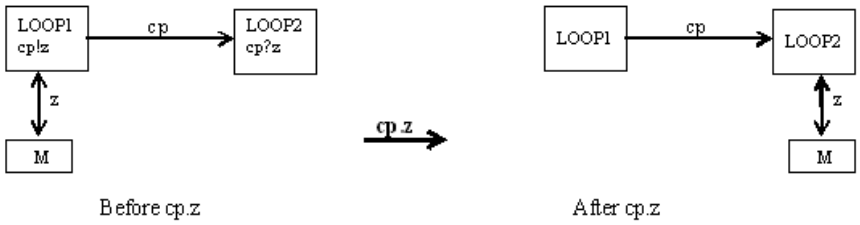

Figure 3: Transfer of machine M through control point cp

the traces of the parallel combination of controllers to be given:

$$traces(LOOP_1 \,||\, \ldots \,||\, LOOP_n) = \{tr \mid \forall 1 \leqslant i \leqslant n \bullet tr \upharpoonright \chi(LOOP_i), mr_i \in traces(LOOP_i)\}$$

Whenever two controllers synchronise on *cp*, the machine reference is passed from one to the other, thus passing control over the associated machine. This is illustrated in Figure 3.

5 Consistency verification

In this section we discuss how divergence freedom of a mobile combined communicating system can be established. We also consider deadlock-freedom.

If the parallel combination of CSP controllers is not divergence free, then the whole system will have divergence. Therefore, the first step is to establish that the CSP part of the system is divergence free. FDR can be used to check divergence freedom of the CSP part of the system by checking the divergence freedom of the parallel combination of controllers. If the CSP part is divergence free, then any divergence in the system must arise from the B machines. Divergence

arises in a machine when its operations are called outside their precondition by the controllers and we are essentially using the divergence freedom check to check this. Thus, the second step in divergence freedom verification is to establish that the operations of all machines in the system are always called inside their precondition by the controllers during the execution.

The key point is that we are allowing B machines to be passed from one controller to another. A controller typically receives a machine from another controller without knowing its state in advance, and so the divergence freedom between a machine and a controller needs to take the combined behaviour of the controllers into account. In order to keep proofs manageable, we need to check the state of the machines when passed from one controller to another, as the target controller does not have any control over the state of the received machine. Therefore, we will need to ensure that a machine is always transferred to another controller in a correct state where its operations will be called appropriately. In order to achieve this, for each control point we assign an assertion on the state of the machine whose reference is passed along that control point. The intention is that whenever a machine reference is passed to a CSP controller along a control point, it is guaranteed that the assertion is satisfied.

The notation $assert(cp_z)$ denotes the assertion of the control point cp for a machine whose reference is z. For instance, $assert(cp_z) : z.n = 0$ means that the variable n of the machine with machine reference z must be zero when passing through cp. For each control point, one assertion is assigned for all machines being passed through it. For a machine with machine reference w, the assertion of cp is $assert(cp_z)$ with w substituted for z which is $w.n = 0$.

To verify that a controller *LOOP* handles the B machines correctly, we translate the body of CSP processes N_i into AMN. We define a translation function, $trans(P_i)$, to translate the body of each process N_i into the corresponding AMN. Verifying $[trans(P)]Q$ will show that the sequence of operations expressed in P will establish the postcondition Q in the B machine, as used later in Definition 2.

Definition 1 *The translation of CSP expressions into AMN is defined as follows:*

$trans(SKIP) = SELECT\ false\ THEN\ skip\ END$

$trans(c?x \rightarrow P(x)) = ANY\ x\ WHERE\ x : type(c)\ THEN\ trans(P(x))\ END$

$trans(c!v \rightarrow P) = PRE\ v : type(c)\ THEN\ skip\ END;\ trans(P)$

$trans(cp?z \rightarrow P(z)) =$ *ANY* z *WHERE* $z : MR$ *THEN* *SELECT* $assert(cp_z)$ *THEN* $trans(P(z))$ *END* *END*

$trans(cp!z \rightarrow P) = PRE\ assert(cp_z)\ THEN\ skip\ END;\ trans(P)$

$trans(z.op!s?t \rightarrow P(t)) = t \longleftarrow z.op(s);\ trans(P(t))$

$trans(P' \,\square\, P'') = CHOICE\ trans(P')\ OR\ trans(P'')\ END$

$trans(P' \sqcap P'') = CHOICE\ trans(P')\ OR\ trans(P'')\ END$

$trans(if\ b\ then\ P'\ else\ P'') = IF\ b\ THEN\ trans(P')\ ELSE\ trans(P'')\ END$

$trans(e \rightarrow P) = skip;\ trans(P)$

$trans(N(v_1, \ldots, v_n)) = rec := N(v_1, \ldots, v_n)$

The last clause introduces a program counter *rec* to handle recursive calls. Observe that inputs $c?x$ and $cp?z$ are translated to the *ANY* statement, which models an assumption that the value being received is of the correct type. In the case of a machine reference, $cp?z$ also contains a

SELECT statement, which models an extra assumption that the machine is in a state satisfying $assert(cp)$. Outputs $c!v$ and $cp!z$ are translated to *PRE* statement rather than *ANY* and *SELECT* statements. v and z are the parameters of the process so there are already some predicates on their value before this stage. Therefore, there is no need to have *ANY* in their translation. Instead, we use the *PRE* statement, which models a guarantee that the condition is met on output values. In the case of a machine reference, there is also no *SELECT* statement in the translation of $cp!z$. This is because we are going to use weakest precondition formulae in our consistency verification strategy and the *PRE* statement is the suitable statement in order to detect when the assertion is not ensured by the sender process, which corresponds to divergence in the system.

Supposing for a process N_i in *LOOP* we can find an invariant referring to all free variables in N_i such that if this invariant is true then whenever N_i is called to be executed, it calls the operations of all the machines it owns at the beginning of that recursive call through their precondition. If we can establish that this invariant holds at every recursive call of N_i, and if the state of the machines N_i receives always satisfy the related assertions, then N_i always calls the operations of all the machines it works with through their precondition at all the time during the execution.

If we can establish the conditions above for each process N_i $(1 \leqslant i \leqslant n)$, then the parallel combination between *LOOP* and any machine it works with during the execution is ensured to be divergence free. As this invariant should be true at each recursive call, we call it Control Loop Invariant, CLI.

We now present the definition below for *LOOP* which contains the conditions we explained above:

Definition 2 *LOOP is called CLI preserver if for each* N_i $(1 \leqslant i \leqslant n)$ *in LOOP, a Control Loop Invariant,* CLI_i*, can be found such that :*

1. $[init_1;\ init_2;\ \ldots;\ init_m;\ rec := N_1](CLI_1)$
2. $\forall 1 \leqslant i \leqslant n \bullet ((rec = N_i \wedge CLI_i) \Rightarrow [trans(P_i)](\forall 1 \leqslant j \leqslant n \bullet (rec = N_j \Rightarrow CLI_j)))$

where $M_1, \ldots, M_m$ *are the machines that LOOP owns at the beginning of the execution and* $init_1, \ldots, init_m$ *are the Initialisation clause of machines* $M_1, \ldots, M_m$ *respectively.*

The theorem below makes use of this definition:

Theorem 1 *Supposing* $LOOP_1, LOOP_2, \ldots, LOOP_n$ *are the CSP controllers and* $M_1, M_2, \ldots, M_m$ *are the B machines in a mobile combined communicating system. If the parallel combination of controllers is divergence free and all controllers are CLI preserver, then the whole system,* $LOOP_1 \parallel LOOP_2 \parallel .... \parallel LOOP_n \parallel z_1 : M_1 \parallel z_2 : M_2 \parallel ... \parallel z_m : M_m$*, is divergence free.*

If each *LOOP* is a CLI preserver then this allows each *LOOP* to be separately checked for divergence-freedom on machines it controls at some point during its execution. This theorem allows all of these individual checks to be combined to an overall consistency result.

We can also establish deadlock-freedom of the overall system by checking deadlock-freedom of the combination of controllers. The B machines do not contribute to any deadlocking behaviour. This is because preconditions do not block, and we are not allowing blocking within the

bodies of operations.

Theorem 2 *Suppose* $LOOP_1, LOOP_2, \ldots, LOOP_n$ *are the CSP controllers and* $M_1, M_2, \ldots, M_m$ *are the B machines in a mobile combined communicating system and the system is divergence free. If the parallel combination of controllers* $LOOP_1 \parallel LOOP_2 \parallel .... \parallel LOOP_n$ *is deadlock free, then the whole system,* $LOOP_1 \parallel LOOP_2 \parallel .... \parallel LOOP_n \parallel z_1 : M_1 \parallel z_2 : M_2 \parallel ... \parallel z_m : M_m$*, is deadlock free.*

One important issue which has been considered in our work is to allow the refinements of the components into our framework. The intention is to be able to substitute a component by its refinement in such a way that the substitution does not have any effect on the system consistency properties. In [Vaj09], it has been proved that if we have a mobile combined communicating system and this system is divergence free and deadlock free, then if we use a refinement of B machines or CSP controllers instead of them in the system, the system remains divergence free and deadlock free. This enables us to use a refinement of a component instead of the component in the system.

6 Case study: Flight tickets sale system

In this section, we present a case study within Mobile $CSP \parallel B$ framework. The case study is a flight tickets sale system which presents the usage of Mobile $CSP \parallel B$ architecture in designing and developing peer to peer networks. We first provide a Mobile $CSP \parallel B$ model of a flight tickets sale system and then we verify the consistency of our model by using theorems 1 and 2. This is a simplified version of the case study to appear in [Vaj09]. It generalises the language presented in Section 3 by introducing a parameter to track a set of machine references. However, the important aspect is that when a machine reference is output then the controller does not retain the machine reference.

This case study is designed as a flight tickets sale system in which tickets of different flights are sold or cancelled. The system contains a Sell agency which sells tickets of different flights to customers, and it contains one Return office which cancels customers' tickets. The Sell agency can only sell flight tickets and it is not able to cancel any tickets of the flights. The Return office is only responsible for cancelling tickets and it is not able to sell any flight tickets.

If the Sell agency or the Return office want to sell or cancel a ticket of a flight, they should have access to the information of that flight. Otherwise they are not able to sell or cancel any tickets. This description of the system makes it clear what should be modeled as the B machines and what should be modeled as the controllers in the system. The B machines in our system are the individual flights. They manage all the booking information of the flights. So each machine represents one of the flights in the system. The Sell agency and the Return office play the role of the controllers in our system.

Each flight machine is given a unique machine reference which is the channel used by the Sell agency and the Return office to contact and communicate with that flight machine. The Sell agency or the Return Office can sell or cancel a ticket of a flight only if they own that machine's reference. If they want to sell or cancel a ticket of a flight but they do not have that machine's reference, they request the machine's reference from each other.

The Sell agency and the Return office behave in such a way so that they do not keep the machines when they can not use them any more. A full machine can not be used any more by the Sell agency as all the tickets have already been sold and there is no more ticket available to be sold next. So, if a flight owned by a Sell agency is full after selling a ticket, the Sell agency passes that full machine to the Return office. On the other hand, an empty machine can not be used any more by the Return office as there is no sold ticket in the machine to be cancelled next. So, if a flight owned by the Return office is empty after cancelling a ticket, the Return office passes that empty machine to the Sell agency. In other words, the Sell agency does not keep full machines with itself and the Return office does not keep empty machines with itself.

A set S is introduced for the Sell agency and for the Return office which contains all the machine references owned by the process. As a result, $SellAgency(S)$ represents the Sell agency which currently owns the machine references in S, and $ReturnOffice(S)$ represents the Return office which currently owns the machine references in S.

For the purposes of our case study, we will use two flight machines: $flight1$ and $flight2$. We also assume that at the beginning of the system execution, the Sell agency owns both flight machines. Therefore, the whole system is as below:

$$ReturnOffice(\{\}) \;||\; SellAgency(\{mr1, mr2\}) \;||\; mr1 : flight1 \;||\; mr2 : flight2$$

where $mr1$ and $mr2$ are the machine references of machines $flight1$ and $flight2$ respectively.

6.1 Design and specification of B machines

Each B machine manages the information of one flight in the system. The structure of all flights in our system are the same so the B machines have the same specification but with different names.

Each B machine contains the information about that particular flight such as: the (positive) number of seats of the flight, and the number of tickets which have already been sold. It also contains some operations for state transitions in the flight such as selling or cancelling tickets and some other operations for finding out the current state of the machine such as whether it is empty or full. Initially, each flight is empty. In other words, no ticket has been sold to anybody at the beginning of the execution. For reasons of space, only the operations of a flight machine are presented here, in Figure 4.

After specifying the flight machines in AMN, we used ProB to verify the internal consistency of our B machines and to explore the behaviour of their operations. As all B machines have the same structure in our system, a single B machine specification was checked, analysed and animated in ProB. The B machine was proved to be internally consistent and it behaved as expected.

6.2 System design and specification in mobile CSP

In this section, we describe the CSP specification of our system according to our mobile architecture. The Return office and the Sell agency are each specified as a CSP process which describes their behaviour in the system. In addition, it is defined who is the controller of each machine at the beginning of the execution.

```
response ← sell(pp) =
  PRE
    pp : Passport  &
    sold ≠ seats
  THEN
    IF pp : Passport − customer
    THEN
      customer := customer ∪ {pp}   ||
      sold := sold + 1   ||
      IF sold + 1 = seats
      THEN response := Full
      ELSE response := Available
      END
    ELSE response := IncorrectInput
    END
  END
```

```
response ← cancel(pp) =
  PRE
    pp : Passport  &
    sold ≠ 0
  THEN
    IF pp : customer
    THEN
      customer := customer − {pp}   ||
      sold := sold − 1   ||
      IF sold − 1 = 0
      THEN response := Empty
      ELSE response := Available
      END
    ELSE response := IncorrectInput
    END
  END
```

```
response ← empty =
  BEGIN
    IF sold = 0
    THEN response := YES
    ELSE response := NO
    END
  END
```

```
response ← full =
  BEGIN
    IF sold = seats
    THEN response := YES
    ELSE response := NO
    END
  END
```

Figure 4: Operations in a flight machine

A function *ref* is used to assign a unique machine reference for each machine in the system. By mapping each machine to a machine reference, the flight machines are given a unique machine reference which then can be used to communicate with the CSP processes. Two control points *dp* and *ep* are introduced for passing the machines between the Sell agency and the Return office. *dp* is a control point channel used to pass the flight machines from the Return office to the Sell agency. *ep* is a control point channel used to pass the flight machines from the Sell agency to the Return office.

The specification of the Sell agency and the Return office is shown in Figures 5 and 6 respectively.

A CSP process, *Controller*, is introduced which is the parallel combination of the Sell agency and the Return office. As we said before, we assume that at the beginning of the system execution, the Sell agency owns both flight machines. As a result, *Controller* is specified as: $Controller = ReturnOffice(\{\}) \;||\; SellAgency(\{mr1, mr2\})$.

The system can be coded for tool analysis into standard CSP, by treating the machine references as data values rather than as channels, and declaring a global channel *MC* (machine channel) which carries machine references as the first value, and then operation names and values as further values. In other words, any call of a machine operation $z.op!s?t$ in the body of the CSP controllers is modelled as $MC.z.op!s?t$ in the standard CSP description of our system.

After coding the system in standard CSP, the behaviour of the Sell agency and the Return office was checked individually by using ProBE. We then used ProBE to explore the execution of *Controller* in order to check their behaviour while working in parallel in the system. In addition,

$$SellAgency(S) = P_1(S)$$

$$P_1(S) = buy?flight?pn \rightarrow if\ ref(flight) \in S \\ then\ P_2(S,flight,pn) \\ else\ P_3(S,flight,pn) \\ \Box \\ ask?flight \rightarrow P_4(S,flight) \\ \Box \\ dp?z \rightarrow P_1(S \cup \{z\})$$

$$P_2(S,f,pn) = ref(f).sell!pn?resp \rightarrow if\ resp = Full \\ then\ P_5(S,f) \\ else\ P_1(S)$$

$$P_3(S,f,pn) = (require!f \rightarrow ((dp?w \rightarrow P_2(S \cup \{w\},f,pn)) \\ \Box \\ (fullMachine \rightarrow P_1(S)))) \\ \Box \\ (ask?flight \rightarrow P_7(S,f,pn,flight)) \\ \Box \\ (dp?w \rightarrow if\ w = ref(f) \\ then\ P_2(S \cup \{w\},f,pn) \\ else\ P_3(S \cup \{w\},f,pn))$$

$$P_4(S,f) = ref(f).empty?resp \rightarrow \\ if\ resp = YES \\ then\ (emptyMachine \rightarrow P_1(S)) \\ else\ (ep!ref(f) \rightarrow P_1(S - \{ref(f)\}))$$

$$P_5(S,f) = (ep!ref(f) \rightarrow P_1(S - \{ref(f)\})) \\ \Box \\ (ask?flight \rightarrow if\ flight = f \\ then\ (ep!ref(f) \rightarrow P_1(S - \{ref(f)\})) \\ else\ P_6(S,f,flight)) \\ \Box \\ (dp?w \rightarrow P_5(S \cup \{w\},f))$$

$$P_6(S,f,flight) = ref(flight).empty?resp \rightarrow \\ if\ resp = YES \\ then\ (emptyMachine \rightarrow P_5(S,f)) \\ else\ (ep!ref(flight) \rightarrow P_5(S - \{ref(flight)\},f))$$

$$P_7(S,f,pn,flight) = ref(flight).empty?resp \rightarrow \\ if\ resp = YES \\ then\ (emptyMachine \rightarrow P_3(S,f,pn)) \\ else\ (ep!ref(flight) \rightarrow P_3(S - \{ref(flight)\},f,pn))$$

Figure 5: Sell agency

Controller was proved to be divergence free and deadlock free by using FDR.

6.3 Verification of the system: divergence-freedom

Our Flight tickets sale system will have divergence if (1) the parallel combination of the Sell agency and the Return office has divergence, or (2) a Sell agency calls the operation *sell* of a full machine during the execution, or (3) the Return office calls operation *cancel* of an empty machine during the execution.

In this section we verify the divergence freedom of our system by using Theorem 1. *Controller* has already been proved to be divergence free by using FDR. The next step is to establish that the Sell agency and the Return office are CLI preserver. In order to achieve this, we should first assign assertions for control points in our system. Then, we should define Control Loop Invariants for the processes in the Sell agency and in the Return office.

If a machine is passed from the Sell agency to the Return office, it should not be empty. On the other hand, if a machine is passed from the Return office to the Sell agency, it should not be already full. *ep* is the control point which passes the machines from the Sell agency to the Return office. So, the assertion of ep should be assigned as $assert(ep_z) : z.sold \neq 0$. *dp* is the control point which passes the machines from the Return office to the Sell agency. So, the assertion of *dp* should be assigned as $assert(dp_z) : z.sold \neq z.seats$

For the Sell agency and the Return office, we can introduce Control Loop Invariants, shown in Figure 7 which establish that both Sell agency and the Return office are CLI preserver. Thus,

$$ReturnOffice(S) = R1(S)$$

$$\begin{aligned}R_1(S) = {} & return?flight?pn \rightarrow if\ ref(flight) \in S \\ & \quad then\ R_2(S,flight,pn) \\ & \quad else\ R_3(S,flight,pn) \\ & \Box \\ & require?flight \rightarrow R_5(S,flight) \\ & \Box \\ & ep?z \rightarrow R_1(S \cup \{z\})\end{aligned}$$

$$\begin{aligned}R_2(S,f,pn) = {} & ref(f).cancel!pn?resp \rightarrow if\ resp = Empty \\ & \quad then\ R_4(S,f) \\ & \quad else\ R_1(S)\end{aligned}$$

$$\begin{aligned}R_3(S,f,pn) = {} & ask!f \rightarrow (ep?w \rightarrow R_2(S \cup \{w\},f,pn) \\ & \quad \Box \\ & \quad emptyMachine \rightarrow R_1(S)) \\ & \Box \\ & require?flight \rightarrow R_7(S,f,pn,flight) \\ & \Box \\ & ep?w \rightarrow if\ w = ref(f) \\ & \quad then\ R_2(S \cup \{w\},f,pn) \\ & \quad else\ R_3(S \cup \{w\},f,pn)\end{aligned}$$

$$\begin{aligned}R_4(S,f) = {} & dp!ref(f) \rightarrow R_1(S - \{ref(f)\}) \\ & \Box \\ & require?flight \rightarrow if\ flight = f \\ & \quad then\ dp!ref(f) \rightarrow R_1(S - \{ref(f)\}) \\ & \quad else\ R_6(S,f,flight) \\ & \Box \\ & ep?w \rightarrow R_4(S \cup \{w\},f)\end{aligned}$$

$$\begin{aligned}R_5(S,f) = {} & ref(f).full?resp \rightarrow \\ & \quad if\ resp = YES \\ & \quad then\ fullMachine \rightarrow R_1(S) \\ & \quad else\ dp!ref(f) \rightarrow R_1(S - \{ref(f)\})\end{aligned}$$

$$\begin{aligned}R_6(S,f,flight) = {} & ref(flight).full?resp \rightarrow \\ & \quad if\ resp = YES \\ & \quad then\ fullMachine \rightarrow R_4(S,f) \\ & \quad else\ dp!ref(flight) \rightarrow R_4(S - \\ & \{ref(flight)\},f)\end{aligned}$$

$$\begin{aligned}R_7(S,f,pn,flight) = {} & ref(flight).full?resp \rightarrow \\ & \quad if\ resp = YES \\ & \quad then\ fullMachine \rightarrow R_3(S,f,pn) \\ & \quad else\ dp!ref(flight) \rightarrow R_3(S - \\ & \{ref(flight)\},f,pn)\end{aligned}$$

Figure 6: Return office

according to Theorem 1 our system is divergence free.

6.4 Verification of the system: deadlock-freedom

By verifying deadlock freedom of our Flight tickets sale system, we establish that there will never occur a situation in which the execution of our system is blocked. This is a natural condition checked for concurrent systems.

The deadlock freedom of our system is verified by using Theorem 2. We have already proved in previous section that the system is divergence free. Finally, *Controller* has already been proved to be deadlock free by using FDR. All conditions in Theorem 2 are true. Thus, our system is deadlock free.

7 Conclusion

In this paper, we introduced Mobile $CSP \parallel B$, a formal framework based on $CSP \parallel B$ which enables us to specify and verify concurrent systems with mobile architecture instead of the previous static architecture. In the previous static version of $CSP \parallel B$, for each operation in a B machine, there is one channel between that machine and its controller. However in our work, a machine's reference is the only channel through which a CSP controller and that B machine can interact with each other. In contrast to static $CSP \parallel B$ in which each controller is dedicated for one B machine, we designed our framework in such a way that controllers are able to work with

$CLI_{P_1(S)} : S \subseteq MR \wedge \forall k \in S \bullet k.sold \neq k.seats$

$CLI_{P_2(S,f,pn)} : CLI_{P_1(S)} \wedge ref(f) \in S \wedge pn \in Passport$

$CLI_{P_3(S,f,pn)} : CLI_{P_1(S)} \wedge ref(f) \in (MR - S) \wedge pn \in Passport$

$CLI_{P_4(s,f)} : CLI_{P_1(s)} \wedge f \in Flights$

$CLI_{P_5(S,f)} : S \subseteq MR \wedge \forall k \in (S - \{ref(f)\}) \bullet k.sold \neq k.seats \wedge ref(f) \in S \wedge ref(f).sold \neq 0$

$CLI_{P_6(S,f,flight)} : CLI_{P_5(S,f)} \wedge flight \in Flights \wedge f \neq flight$

$CLI_{P_7(S,f,pn,flight)} : CLI_{P_3(S,f,pn)} \wedge flight \in Flights$

$CLI_{R_1(S)} : S \subseteq MR \wedge \forall k \in S \bullet k.sold \neq 0$

$CLI_{R_2(S,f,pn)} : CLI_{R_1(S)} \wedge ref(f) \in S \wedge pn \in Passport$

$CLI_{R_3(S,f,pn)} : CLI_{R_1(S)} \wedge ref(f) \in (MR - S) \wedge pn \in Passport$

$CLI_{R_4(S,f)} : S \subseteq MR \wedge \forall k \in (S - \{ref(f)\}) \bullet k.sold \neq 0 \wedge ref(f) \in S \wedge ref(f).sold = 0$

$CLI_{R_5(S,f)} : CLI_{R_1(S)} \wedge f \in Flights$

$CLI_{R_6(S,f,flight)} : CLI_{R_4(S,f)} \wedge flight \in Flights \wedge flight \neq f$

$CLI_{R_7(S,f,pn,flight)} : CLI_{R_3(S,f,pn)} \wedge flight \in Flights$

Figure 7: Control Loop Invariants in the Sell agency and the Return office

different machines during the execution. Controllers exchange machines between each other by exchanging the machine references. This results from two facts about our system architecture:

1. In Mobile $CSP \parallel B$, we have introduced mobile channels: machine references are mobile channels and they can move around the system during the execution.

2. In Mobile $CSP \parallel B$, mobile channels (machine references) are allowed to be passed between processes through static channels (control points) in the system.

In addition, we defined and verified the conditions which guarantee the divergence freedom and deadlock freedom consistency of the systems specified and designed in Mobile $CSP \parallel B$.

The case study demonstrates the applicability of our mobile $CSP \parallel B$ framework in specifying and verifying mobile communication systems. It shows that the theorems are sufficient to manage concurrent updates of state. It also demonstrates the ability of the CSP controllers to interact with numerous machines at the same time.

Apart from formal method integrations, some other approaches have also been created in modelling and verifying mobile systems one of which is Mobile UNITY. Mobile UNITY [MR96], an extension of the parallel program design language UNITY [CM88], is a language and proof logic for specifying and reasoning about concurrent mobile systems. In Mobile UNITY, components can move around and execute at different locations and they can interact and communicate with each other during the execution. In Mobile UNITY notation, mobility is modelled as the change of the location of components. In other words, the change in the location of components provides means to model movement in the system. It allows the description of location-sensitive behaviour, e.g., interaction at the same place, or within a certain distance. Each component in Mobile UNITY has a distinguished location variable. The location of each component is modelled by assignment of a value to its location variable. The movement of a component is modelled by the change of value of its location variable. Mobile UNITY proof logic is employed to verify the safety and liveness properties of a system expressed in the Mobile UNITY notation. Mobile UNITY has no notion of refinement.

We believe using CSP language in our approach makes flow of control more explicit in contrast to approaches that use control variables. In addition, Mobile *CSP* || *B* framework supports a refinement approach: it enables the refinements of components to be a substitute of the original components in the system while the system consistency is guaranteed to remain. Furthermore, as our framework can be coded into the original constructs of CSP and B, we are able to use the comprehensive and highly efficient software tools of CSP and the B-Method to analyse, animate and check the behaviour and the consistency of the two parts of a mobile system separately. This shows the advantage of using B-Method as our chosen state based formal method and CSP as the process algebra in our framework.

Our approach uses syntactic restrictions on process terms to ensure that at most one controller is in possession of a machine reference at any one time. This approach imposes certain restrictions on the language to achieve the required result, for example to do with the careful handling and restrictive use of machine references as process parameters; and avoidance of internal parallelism within controllers. An alternative approach [Gol09] would be to use a 'monitoring process' for a CSP controller process which tracks which machine references the process is supposed to have, and checks that it does not make use of references it should not have. Such monitoring processes can be used within FDR checks to ensure that controllers behave in the required way.

The approach taken in $\pi \mid B$ [KST07] also allows dynamic creation of machines and channels, and reconfiguration of the network. In contrast, the approach taken in this paper provides a more static network between the controllers, but developing the framework to enable reconfiguration, and dynamic process and machine creation, would be an interesting avenue of future research.

Acknowledgements: We are grateful to Michael Goldsmith for comments and discussions on various aspects of this paper that have improved our understanding. We are also grateful to the anonymous reviewers for their comments.

Bibliography

[Abr96] J. R. Abrial. *The B Book: Assigning Programs to Meaning*. CUP, 1996.

[BC02] B-Core. B-Toolkit. 2002.
http://www.b-core.com/btoolkit.html

[But00] M. Butler. csp2B: A Practical Approach To Combining CSP and B. *Formal Aspects of Computing* 12, 2000.

[Cle09] ClearSy. Atelier B 4.0. 2009.
http://www.atelierb.eu/index-en.php

[CM88] K. M. Chandy, J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[FSEL07a] Formal Systems (Europe) Ltd. FDR 2.83 Manual. 2007.
http://www.fsel.com

[FSEL07b] Formal Systems (Europe) Ltd. ProBE. 2007.
http://www.fsel.com

[Fis97] C. Fischer. CSP-OZ: A combination of Object-Z and CSP. In *FMOODS '97*. 1997.

[Gol09] M. Goldsmith. Personal communication, 28th October. 2009.

[GS97] A. J. Galloway, W. J. Stoddart. An operational semantics for ZCCS. *ICFEM '97*, 1997.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[KST07] D. Karkinsky, S. Schneider, H. Treharne. Combining mobility with state. *IFM'07*, 2007.

[LB03] M. Leuschel, M. Butler. ProB: A Model Checker for B. In *FM 2003*. 2003.
http://www.stups.uni-duesseldorf.de/ProB/overview.php

[Mor90] C. C. Morgan. Of wp and CSP. *In W.H.J. Feijen, A. J. M. van Gesteren, D. Gries, and J. Misra, editors, Beauty is our business: a birthday salute to Edsger W. Dijkstra. Springer-Verlag*, 1990.

[MR96] P. J. McCann, G. C. Roman. Mobile UNITY: A language and logic for concurrent mobile systems. *Technical Report WUCS-97-01, Department of Computer Science, Washington University in St. Louis*, 1996.

[OCW07] M. V. M. Oliveira, A. L. C. Cavalcanti, J. C. P. Woodcock. A UTP Semantics for *Circus*. *Formal Aspects of Computing* 21, 2007.

[Ros08] A. Roscoe. On the expressiveness of CSP. 2008. Draft of October 23, 2008.

[Sch00] S. Schneider. *Concurrent and Real-time Systems: the CSP approach*. Wiley, 2000.

[Sch01] S. Schneider. *The B-method: an introduction*. Palgrave Macmillan, 2001.

[ST05] S. Schneider, H. Treharne. CSP Theorems for Communicating B machines. *Formal Aspects of Computing* 17, 2005.

[TDC04] K. Taguchi, J. Dong, G. Ciobanu. Relating pi-calculus to Object-Z. *ICECCS*, 2004.

[TS02] H. Treharne, S. Schneider. Communicating B machines. *ZB2002*, 2002.

[Vaj09] B. Vajar. *Mobile CSP||B*. PhD thesis, University of Surrey, in preparation, 2009.

[VST07] B. Vajar, S. Schneider, H. Treharne. Introducing Mobility into CSP||B. In *AVOCS 2007*. 2007.