How to drive a B Machine

Helen Treharne and Steve Schneider

Department of Computer Science, Royal Holloway,
University of London, Egham, Surrey, TW20 0EX, UK.

E-mail: {helent,steve}@dcs.rhbnc.ac.uk

Fax: +44 (0)1784 439786

Abstract. The B-Method is a state-based formal method that describes behaviour in terms of *MACHINES* whose states change under *OPERATIONS*. The process algebra CSP is an event-based formalism that enables descriptions of patterns of system behaviour. We present a combination of the two views where a CSP process acts as a control executive and its events simply drive corresponding *OPERATIONS*. We define consistency between the two views in terms of existing semantic models. We identify proof conditions which are strong enough to ensure consistency and thus guarantee safety and liveness properties.

Keywords: B-Method, CSP, Embedded Systems, Programming Calculi, Combining Formalisms.

1 Introduction

State based methods such as B specify functional aspects of a system and the effect of individual operations. On the other hand event-based process algebras are concerned with patterns of operations. System designers are interested in both these aspects of a system and thus a combination of state and event based descriptions of a system is desirable. The systems that originally motivated our need to consider both viewpoints were safety-critical systems, for example embedded interlock systems. This paper provides a safe way of describing a combined view of a system.

Systems have successfully been modelled as collections of interdependent machines within the B Method. An abstract *MACHINE* is described using the Abstract Machine Notation (AMN). In this paper we adopt the convention that AMN keywords are indicated in italic capitals. Large *MACHINE*s can be constructed from other *MACHINE*s using *INCLUDES*, *SEES* and other constructs. A *MACHINE* encapsulates some local state and a collection of modules called *OPERATIONS*. *OPERATIONS* in a *MACHINE* can be pre-conditioned or guarded. We are interested in specifying embedded systems and refer to a B Abstract System in terms of the *MACHINE* at the top of the hierarchy of *MACHINES* which specify the following two kinds of *OPERATIONS*. Firstly, pre-conditioned *OPERATIONS* describe the modules which will be refined to code. They have the form *PRE R THEN T END*. If an *OPERATION* is invoked when the pre-condition R is true it will behave as specified by T. However, if the *OPERATION* is invoked outside its pre-condition the resulting execution may be an incorrect behaviour of the system. Secondly, the *OPERATIONS* which provide a model of the system context have the form *SELECT P THEN V END*, where P is a guard and V describes the effect of invoking the *OPERATION*. Guards are predicates on the state of a *MACHINE* which constrain the cases when an *OPERATION* is entitled to be invoked. If an *OPERATION* is invoked when the guard is true then the system will behave as expected with respect to the specification as was the case above. However, if the guard is false then execution is blocked.

Process algebras such as Communicating Sequential Processes (CSP) [8] are concerned with the evolution of systems as they execute sequences of events. They are appropriate for describing execution patterns. In this paper, we will show how events in a CSP recursive loop determine which corresponding *OPERATION* should execute. Thus we view the AMN specifications as providing abstract models of reactions to events. The recursive loop can be viewed as an execution checker and we will refer to it as a *control executive*. Thus in a combined view of a system a control executive for a system is described using a process algebra which in turn drives the individual state transitions of an Abstract System.

In general a CSP control executive could invoke an *OPERATION* outside its pre-condition, resulting in divergent behaviour. In [15] we gave conditions which ensured this did not occur. With guarded *OPERATIONS* we also need to ensure deadlock freedom so that a control executive never gets stuck trying to invoke *OPERATIONS* which are blocked. Ensuring deadlock freedom is the contribution of this paper.

The main result of this paper is that we introduce a new proof condition which guarantees deadlock freedom in the context of divergence freedom. Furthermore, we verify that this new condition is strong enough to ensure the consistency of a combined system consisting of guarded *OPERATIONS*. In this verification we think of an Abstract System as a process and its combination with the control executive is essentially their parallel composition in CSP.

In formally justifying the link between these state and event-based methods we were influenced by the existing correspondence between Action Systems and CSP. This correspondence is described by Morgan [9] in terms of weakest precondition semantics and the failures-divergences model.

We assume the reader is familiar with AMN. Further details can be found in [1]. However, we will introduce the CSP notation we require. This paper is organised as follows. Section 2 gives a brief overview of CSP. Sections 3, 4 and 5 contain the main contribution of the paper. They present the theoretical foundations of the specific relationship between B and CSP. Section 6 illustrates this new relationship in relation to our previous work on divergence freedom.

The final section contains a discussion and conclusions. Proofs of the results have been omitted for reasons of space and can be found in the technical report [16].

2 Overview of CSP

This section provides a brief introduction to the CSP used in this paper. More details can be found in [8,13,11].

2.1 The language

CSP describes systems in terms of *processes*, which perform *events*. The set of all events is called Σ. Events are either atomic (e.g. *on*, *off*), or they may be structured into a number of components separated by dots (e.g. *send*.5). Communications of values along channels will be described using structured events, so for example the transmission of value 5 along channel *send* will be described with the event *send*.5.

CSP provides a language for describing processes. This includes basic processes such as $STOP$, the process which does nothing, and DIV, the (divergent) process which represents an infinite internal loop. It also contains process constructors for building up process descriptions. The *event prefixing* expression $a \rightarrow P$ means that the process is prepared to engage in the atomic event a and then behaves as the process P. Input of a value x of type T along a channel c is described by $c?x : T \rightarrow P$, where P is the subsequent process, which may depend on the input value x. Output of a value v along channel c is described as $c!v \rightarrow P$ with the subsequent behaviour given by P.

The expression $P \mathbin{\Box} Q$ offers an *external choice* between the two processes P and Q: initially it is prepared to behave as either P or Q, and this choice is resolved by the occurrence of the first event, which can be chosen by the user or environment of this choice. Standard conditional statements *if* b *then* P *else* Q are also in the language. Processes execute in parallel by requiring synchronisation on events. The parallel combination $P \parallel Q$ executes P and Q concurrently, but the combination can only perform an event when both parties are willing to perform it. Thus parallel combination can introduce *deadlock* if the parties cannot agree on any next event.

Finally, processes can be defined by means of recursive definitions: the names of recursively defined processes can be referred to in the process definitions themselves. For example, a one place buffer containing a value v can be defined as follows:

$$BUF(v) = out!v \rightarrow in?x : T \rightarrow BUF(x)$$

The process definition on the right hand side is called the body of the definition. In fact this is a family of equational definitions, one for each possible value of v. The family of processes defined in this way could also be written as a *vector* of processes $\underline{BUF}$ indexed by the possible values that v could take.

2.2 Semantics

The CSP approach to semantics is to define the semantics of a process as the set of all observations that may be made of it. The particular kind of observation determines the semantic model being used. All the models have a structure which ensures that recursively defined processes are always well-defined.

The simplest model is the *traces* model which describes processes P in terms of $traces(P)$, the set of all possible sequences of events that P can perform. The CSP process operators are such that the traces of a process can be determined in a compositional way from the traces of its components, so for example

$$traces(a \rightarrow P) = \{\langle\rangle\} \cup \{\langle a \rangle \frown tr \mid tr \in traces(P)\}$$

Here $\langle\rangle$ is the empty sequence, and $tr_1 \frown tr_2$ is the concatenation of tr_1 and tr_2.

Another semantic model is the *stable failures* model. In this model the semantic value of a process P is given as two sets: the set of *traces* as in the previous model, and the set of all its *stable failures*. A stable failure of a process P is a trace/refusal pair (tr, X) that P can exhibit, where tr is a sequence of events that P can perform, reaching a stable state (from which no further internal progress can occur), and $X \subseteq \Sigma$ is a set of events that P can *refuse* to participate in from that state. Thus the process DIV has $traces(DIV) = \{\langle\rangle\}$, and no stable failures at all (since it never reaches a stable state). Examples of stable failures of $BUF(3)$ include the empty failure $(\langle\rangle, \varnothing)$ and the trace/refusal pair $(\langle out.3, in.6\rangle, \{out.4, out.5\})$.

The stable failures model is unable to address the issue of divergence, since it does not contain any information about unstable states. Instead a third more complicated semantic model, the *failures-divergences* model, provides a suitable treatment of divergence. The semantic value of a process in this model consists of two sets: its set of *divergences*—sequences of events which can lead to infinite internal progress; and its set of all failures, which comprises the stable failures together with (for technical reasons) all divergence/refusal pairs. For divergence-free processes, the stable failures model and the failures-divergences model are equivalent. In this paper we will use the stable failures model in this context for simplicity.

2.3 Specification

A specification is a predicate on all of the possible behaviours of a process. We will be concerned with specifications on traces (written $S_T(tr)$), and specifications on stable failures (written $S_F(tr, X)$). A process P meets a trace specification $S_T(tr)$ if

$$P \textbf{ sat } S_T(tr) \Leftrightarrow \forall\, tr \in traces(P) \bullet S_T(tr)$$

Trace specifications are used to capture safety requirements on processes, i.e. they constrain which traces may occur. For example, the trace specification $S_B(tr) = tr \downarrow in \leqslant tr \downarrow out$ states explicitly that the number of communications on channel *in* should be no greater than the number on *out* (we use the notation $tr \downarrow c$ to denote the number of communications on c appearing in the trace tr). Thus $BUF(3)$ **sat** $S_B(tr)$.

Similarly:

$$P \textbf{ sat } S_F(tr, X) \Leftrightarrow \forall (tr, X) \in \mathit{stable\ failures}(P) \bullet S_F(tr, X)$$

We can use refusal sets of a stable failure to describe liveness requirements. For example, the requirement that a process should be deadlock-free is expressed with the predicate '$X \neq \Sigma$'—that the refusal set X should never be the set of all events. This follows from the fact that a system has reached deadlock precisely when it can make no further progress—that it refuses to perform any more events, corresponding to a possible refusal set of Σ. Deadlock freedom requires that this can never occur. Thus $BUF(3)$ **sat** $X \neq \Sigma$.

An assertion of the form P **sat** S can be established by considering the semantics of P in the appropriate model. There are specification proof rules for each operator derived from the rules used in defining the semantics. There is also a rule for establishing when vectors of processes $\underline{N}$ defined by mutual recursion $\underline{N} = \underline{F}(\underline{N})$ meet (pointwise) a corresponding vector of satisfiable specifications $\underline{S}$. The general form of the rule is as follows:

$$\frac{\forall \underline{Y} \bullet \underline{Y} \textbf{ sat } \underline{S} \Rightarrow \underline{F}(\underline{Y}) \textbf{ sat } \underline{S}}{\underline{N} \textbf{ sat } \underline{S}}$$

This rule works both within the traces model (with each S as a trace specification) and in the stable failures model (with each S as a specification on stable failures). For example, in the traces model the rule can be used to show that $\underline{BUF}$ **sat** $\underline{S_B(tr)}$.

3 A simple coupling between B and CSP loops

In this section and in Sections 4 and 5 we define and verify the framework so that a control executive, containing CSP events, ensures that the guards of the corresponding B *OPERATIONS* are enabled. The *OPERATIONS* model the reaction to the events in the control executive. Introducing the framework in stages aids clarity of presentation and the appropriate development of technical details. We have already stated that the applications that originally motivated this work were safety-critical systems. These systems are designed to run on sequential processors so in this paper we are not concerned with concurrency issues.

This section is split into two parts. Firstly, we discuss how a control executive can be described. Secondly, we discuss consistency between a control executive and an Abstract System. In doing so, we briefly review previous work on control executives and proof conditions which ensure divergence freedom of these loops and their associated Abstract Systems. Then, we identify a new condition to show when control executives are consistent with *MACHINE* descriptions which contain guarded *OPERATIONS*, in the sense that they do not introduce unexpected deadlocks.

3.1 Developing a Control Executive

Consider a recursive CSP process, *LOOP*. In general this will be defined using a parameterised mutual recursion. A family of processes $S(p)$ is used to define *LOOP*, where p is a collection of parameters for keeping track of which process to execute. In the *BUF* example of Section 2.1, the process *BUF* was parameterised by the contents of the buffer v. Each process definition of a control executive represents a sequence of B *OPERATIONS* to be executed by using an event E_{op} for each B *OPERATION* op. Only information which affects the execution of the *OPERATIONS* needs to be carried in the parameters. In the simple case they will simply be numerical indices. For example, the following *LOOP* describes a recursive process which alternates between the events E_{up} and E_{down}.

$$\begin{aligned} LOOP &= S(0) \\ S(0) &= E_{up} \rightarrow S(1) \\ S(1) &= E_{down} \rightarrow S(0) \end{aligned}$$

So in general we would have the following

$$\begin{aligned} LOOP &= S(0) \\ S(0) &= R_0 \\ &\vdots \\ S(n) &= R_n \end{aligned}$$

where *LOOP* is bound to a process name with an initial parameter of 0 and each R_i is a CSP process expression which will describe some behaviour of the *OPERATIONS* and the possible $S(i)$s that can subsequently be reached.

We first introduced the syntax of our control language in [15] to develop non-terminating loops. In this paper, our syntax will also enable us to define terminating loops. However, as we stated above the framework will be developed in stages. Thus we start with simple non-terminating loops consisting of atomic events. The syntax of the CSP terms in the process bodies for non-terminating loops is given by the following pseudo-BNF rule:

$$R ::= a \rightarrow R \mid R_1 \;\Box\; R_2 \mid S(p)$$

Event prefixing and external choice are defined as in Section 2. The event a is of the form E_{op} where op is a B *OPERATION*. $S(p)$ is a process name where p is an expression. Each process body will contain a recursive call, $S(p)$. For example, in the above process $S(1)$ the last term in its definition is $S(0)$ in order to provide a binding for the mutual recursive case. Furthermore, we restrict $S(p)$ from being part of a choice. This restriction is convenient for technical reasons which will be elaborated in Section 3.5.

```
MACHINE embedded_switch
VARIABLES person , status
INVARIANT person ∈ ℕ ∧ status ∈ 0 .. 1 ∧ person ≤ 5
INITIALISATION person := 0 || status := 0

OPERATIONS
  on ≙ SELECT person > 0 ∧ status = 0 THEN
      status := 1
    END ;
  off ≙ SELECT person = 1 ∧ status = 1 THEN
      status := 0
    END ;
  enter ≙ SELECT person < 5 THEN
      person := person + 1
    END ;
  leave ≙ SELECT person > 0 THEN
      person := person − 1
    END
END
```

Fig. 1. Embedded light switch

3.2 Consistency of a CSP Control Executive and a B Abstract System

Once we have a CSP control executive, *LOOP*, we will need to demonstrate that it is appropriate for a particular Abstract System, *M*, by defining the notion of deadlock freedom on the combination ($LOOP \parallel M$). Abstract Systems can be given CSP failures-divergences semantics as shown in [9,3].

Deadlock can occur in a B Abstract System when the guard of an *OPERATION* is false and thus execution is disallowed. Therefore, in our correspondence between CSP and B, the notion of deadlock freedom we require is that not all of the *OPERATIONS* offered in the CSP are actually blocked (with false guards) in the B.

Consider the *MACHINE* in Figure 1. It defines an embedded light switch in a room. A simple control executive for this *MACHINE* allowing only one person in the room at a time would be:

$$ROOM = S(0)$$
$$S(0) = E_{enter} \rightarrow E_{on} \rightarrow E_{off} \rightarrow E_{leave} \rightarrow S(0)$$

Clearly all the guards of the *OPERATIONS* are true whenever they are invoked by the control executive. Conversely, if we tried to turn the light off when two people are in the room it would deadlock, since the value of *person* does not match the guard.

3.3 Reviewing Divergence Freedom

Recall that in Section 3.1 we referred to a family of processes $S(p)$ to define a mutually recursive process *LOOP*. In [15] we used such recursive loops to control the execution of pre-conditioned *OPERATIONS*. In order to ensure consistency of sequences of *OPERATIONS* of the form *PRE R THEN W END* (where *W* did not contain guarded substitutions) we needed to find a *control loop invariant*, *CLI*.

In [15] we stated that the *CLI* need not hold after each individual *OPERATION* but must hold at every recursive call in order to guarantee divergence freedom. Reaching a recursive call corresponds to a maximal trace of a body of a process expression. We also introduced two conditions. Firstly, the initialisation of the Abstract System establishes the *CLI*. Secondly, any sequence of *OPERATIONS*, corresponding to execution between recursive calls also establishes the *CLI*. We then defined the notion of consistency so that if such sequences of *OPERATIONS* could establish the *CLI* we knew that all the *OPERATIONS* were called within their pre-conditions and terminated. If we could demonstrate this for all the bodies of a mutually recursive loop then the loop was demonstrated to be divergence-free.

3.4 Conditions for deadlock freedom

In this paper we will also use the above *CLI* to record that we are at a recursive call of a control executive. In essence this will serve as an anchor for examining the *MACHINE*'s possibilities at each point through the processes of a control executive. We need a stronger invariant than the invariant of an Abstract System. A *CLI* which is appropriate for the process *ROOM* and the *embedded_switch* Abstract System is $person = 0 \wedge status = 0$. The importance of the *CLI* for this example is highlighted at the end of Section 3.5.

The nature of what we have to prove here is stronger than in [15]. However, all of the analysis done in the following sections is done in the context of divergence freedom and the presence of a *CLI* so the stable failures model for CSP will be sufficient for our needs. We need to make sure that we do not deadlock at any point during the execution of the processes and so all the traces along the bodies of processes need to be checked individually. In this paper we define a function *PAIRS* to relate CSP traces to *MACHINE* guards. We show that the following condition is sufficient to establish deadlock freedom for simple non-terminating loops since $traces(DIV) = \{\langle\rangle\}$, any trace of $R_p[\underline{DIV} \,/\, \underline{S}]$ must be a trace of the body of R_p.

Condition 1 $\forall\, tr \bullet tr \in traces(\, R_p[\underline{DIV} \,/\, \underline{S}]\,) \wedge$
$CLI \wedge I \wedge c_b = p \Rightarrow wp(tr, PAIRS(tr, R_p))$

This condition states that for all traces, tr, of R_p before a recursive call and given the invariants hold before the body is executed with the appropriate value of the control variables, the state reached after that trace enables a guard of at least one of the *OPERATIONS* corresponding to the next possible events. In

$$\begin{aligned}
PAIRS(\langle\rangle, E_a \rightarrow R) &= g_a \\
PAIRS(\langle E_a\rangle \frown tr, E_a \rightarrow R) &= PAIRS(tr, R) \\
PAIRS(\langle\rangle, R \,\Box\, R') &= PAIRS(\langle\rangle, R) \vee PAIRS(\langle\rangle, R') \\
PAIRS(tr, R \,\Box\, R') &= \begin{cases} PAIRS(tr, R) \;if\; tr \notin traces(R') \\ PAIRS(tr, R') \;if\; tr \notin traces(R) \\ PAIRS(tr, R) \\ \quad \wedge PAIRS(tr, R') otherwise \end{cases} \\
&\qquad\qquad\qquad\qquad \text{where } tr \neq \langle\rangle \\
PAIRS(tr, S(p')) &= true
\end{aligned}$$

Fig. 2. *PAIRS* definition

fact whenever $tr \in traces(R_p[\underline{DIV} \,/\, \underline{S}])$ the function $PAIRS(tr, R_p)$ gives the weakest condition which needs to be true so that some *OPERATIONS* will be enabled by R_p after tr has occurred.

Each process body R_p is subscripted with p to highlight which process is referred to within the family of processes and its body is bound by a process name $S(p)$, as stated in Section 3.1. The predicate $c_b = p$ arises from modelling control variables to correspond to which process $S(p)$ is being executed. The control variables are not part of the Abstract System but do correspond to AMN variables which is why they are subscripted with b. There will be one control variable for each CSP parameter and thus one corresponding predicate. The value of c_b equals the value of the index in the parameter of the process. It is present because the *CLI* could relate the parameters of the processes with the state of the B *MACHINE*.

For each process body R_p of a particular control system this condition gives rise to several proof obligations that would need to be proved. In the above condition we extract the traces of the body of a process when we view R_p as a function and substitute the process *DIV* for the appropriate recursive call S ($R_p[\underline{DIV} \,/\, \underline{S}]$). We use the process *DIV* since it is the base case in the CSP stable failures model when building up the traces for the process body, i.e. its only trace is the empty trace. For example, given a process body $R_0 = E_b \rightarrow ((E_c \rightarrow S(0)) \,\Box\, (E_d \rightarrow S(0)))$ only $\langle\rangle$, $\langle E_b\rangle$, $\langle E_b, E_c\rangle$ and $\langle E_b, E_d\rangle$ need to be checked. These are the only traces of $E_b \rightarrow ((E_c \rightarrow DIV) \,\Box\, (E_d \rightarrow DIV))$.

3.5 Determining guards using *PAIRS*

In Condition 1 we introduced the function *PAIRS* given in Figure 2. Given a particular sequence of events and a CSP process body the function determines which corresponding guards in the B should be offered next. It is defined over the terms in our CSP language and their trace semantics. In the definition of *PAIRS* the guard of an *OPERATION op* is denoted by g_{op}.

For the term $E_a \rightarrow R$, if the trace is empty then the guard from the corresponding *OPERATION* a is offered. Therefore, if we had a process of this form we would have to check that the $CLI \wedge I \wedge c_b = p \Rightarrow wp(\langle\rangle, g_a)$ in Condition 1 holds in the first instance, i.e. the invariants and the control predicate must be strong enough to imply the guard of the *OPERATION* a. If the trace, tr, is not empty then the function $PAIRS(tr, R)$ represents the disjunction of all the guards of all the *OPERATIONS* of the B *MACHINE* that the CSP control executive might perform next.

The definition containing the external choice term reflects the fact that when the trace is empty the choice is not resolved so either $PAIRS(\langle\rangle, R)$ or $PAIRS(\langle\rangle, R')$ holds so that at least one path of a process containing a choice will not deadlock. On the other hand, when the trace is not empty and the trace is of both R and R' their conjunction must hold since the CSP control could be behaving as either R or R'. Therefore, the B *MACHINE* should be able to respond in both cases. Thus both possibilities should be deadlock-free. If the trace is of either R or R' but not both, then deadlock freedom is required only for the appropriate branch of the choice.

The case containing the recursive call, $S(p')$, gives the predicate $true$. This is required as a base case. By the time a recursive call is reached, the existence of a CLI already ensures that the body of the loop is guaranteed to terminate and nothing further needs to be proved. For example, given the following process within a family of processes $P(0) = E_a \rightarrow E_b \rightarrow P(1)$ and the maximal trace of its body, $\langle E_a, E_b\rangle$, the clause for the recursive case contributes to a simple instance of Condition 1 where the maximal trace terminates, i.e. $CLI \wedge I \wedge c_b = 0 \Rightarrow wp(\langle E_a, E_b\rangle, PAIRS(\langle E_a, E_b\rangle, E_a \rightarrow E_b \rightarrow P(1))) = wp(\langle E_a, E_b\rangle, true)$.

In Section 3.1 we restricted the binding term from being part of a choice. If we had allowed $S(p')$ to be part of a choice we would have had to provide a more complex translation mapping which referred to the guard of the first event of the next process, $R_{p'}$, to be executed. Thus at the cost of reduced expressiveness we prefer to restrict how $S(p')$ can be used.

In practice this restriction does not cause a problem, since such choices can be re-written. For example,

$$\begin{aligned} S(0) &= E_{up} \rightarrow S(2) \\ S(2) &= S(0) \;\Box\; (E_{down} \rightarrow S(0)) \end{aligned}$$

can be re-written as

$$\begin{aligned} S(0) &= E_{up} \rightarrow S(1) \\ S(1) &= (E_{up} \rightarrow S(1)) \;\Box\; (E_{down} \rightarrow S(0)) \end{aligned}$$

whose behaviour in any case is easier to understand.

For the example in Figure 1 with control executive *ROOM* Condition 1 gives rise to the following checks that we have to prove:

$$CLI \wedge I \wedge c_1 = 0 \Rightarrow wp(\langle\rangle, g_{enter}) = g_{enter} \tag{1}$$
$$CLI \wedge I \wedge c_1 = 0 \Rightarrow wp(\langle E_{enter}\rangle, g_{on}) \tag{2}$$
$$CLI \wedge I \wedge c_1 = 0 \Rightarrow wp(\langle E_{enter}, E_{on}\rangle, g_{off}) \tag{3}$$
$$CLI \wedge I \wedge c_1 = 0 \Rightarrow wp(\langle E_{enter}, E_{on}, E_{off}\rangle, g_{leave}) \tag{4}$$
$$CLI \wedge I \wedge c_1 = 0 \Rightarrow wp(\langle E_{enter}, E_{on}, E_{off}, E_{leave}\rangle, true) \tag{5}$$

where $CLI = person = 0 \wedge status = 0$ and

$$I = person \in \mathbb{N} \wedge status \in 0..1 \wedge person \leq 5)$$

The above proof obligations highlight the importance of the *CLI*. The invariant of the *MACHINE* alone is not strong enough to imply the guard of the B *OPERATION enter*, i.e. $person < 5$. In other words, there are some states in which *enter* is blocked. The *CLI* is used to establish that whenever *enter* is called by the control executive it is not blocked. All these obligations are trivial to prove. For example,

$$\begin{aligned} person = 0 \wedge status = 0 \wedge \\ person \in \mathbb{N} \wedge status \in 0..1 \wedge person \leq 5 \wedge \\ c_1 = 0 \Rightarrow \\ [person \Longrightarrow person := person + 1] person > 0 \wedge status = 0 \end{aligned}$$

From the above proof obligations you will notice that we are abusing the *wp* notation. There is one-to-one mapping between the CSP events and their corresponding B *OPERATIONS*. We could set up a formal correspondence to capture this notion, where the empty trace $\langle\rangle$ corresponds to *skip*, the singleton trace $\langle E_a\rangle$ is simply the *OPERATION* named a, $\langle E_a, E_b\rangle = a; b$ and so on.

3.6 Verification of deadlock freedom consistency

Condition 1 in Section 3.4 is sufficient to ensure consistency between which *OPERATIONS* can be executed in the B and what is allowed by the CSP control executive. Therefore when the condition is met, *LOOP* is appropriate for the Abstract System M as stated below in Theorem 1. We need the following lemmas in order to prove the theorem.

The first lemma links the refusals of a CSP process with the guards of its corresponding *OPERATIONS* by use of the function *PAIRS*. The lemma formalises that the state reached after a trace tr ensures a guard of the possible next events, i.e. after tr, it is guaranteed that some *OPERATION* x *not* in the refusal set X has its guard g_x true. The information for the refusals comes from the CSP stable failures semantics.

Lemma 1. *If R is a process body then*
$(tr, X) \in failures(R) \wedge wp(tr, PAIRS(tr, R)) \Rightarrow wp(tr, \bigvee_{x \in \Sigma - X} g_x)$

In Section 3.2 we defined the control executive *ROOM*. Consider the example of the singleton trace $\langle E_{enter} \rangle$ with subsequent refusal $\{E_{enter}, E_{leave}, E_{off}\}$. The above lemma allows us to conclude that the following holds:

$$
\begin{aligned}
&(\langle E_{enter} \rangle, \{E_{enter}, E_{leave}, E_{off}\}) \\
&\in \mathit{failures}(E_{enter} \rightarrow E_{on} \rightarrow E_{off} \rightarrow E_{leave} \rightarrow S(0)) \wedge \\
&\mathit{wp}(\langle E_{enter} \rangle, \mathit{PAIRS}(\langle E_{enter} \rangle, E_{enter} \rightarrow E_{on} \rightarrow E_{off} \rightarrow E_{leave} \rightarrow S(0))) \\
&\qquad\qquad\qquad \Rightarrow \mathit{wp}(\langle E_{enter} \rangle, \bigvee_{x \in \Sigma - X} g_x)
\end{aligned}
$$

In this case $\Sigma = \{E_{enter}, E_{leave}, E_{on}, E_{off}\}$ so $\bigvee_{x \in \Sigma - X} g_x = g_{on}$, and this instance of Lemma 1 reduces to the following which is true.

$$
\begin{aligned}
&(\langle E_{enter} \rangle, \{E_{enter}, E_{leave}, E_{off}\}) \\
&\in \mathit{failures}(E_{enter} \rightarrow E_{on} \rightarrow E_{off} \rightarrow E_{leave} \rightarrow S(0)) \wedge \\
&\mathit{wp}(\langle E_{enter} \rangle, g_{on}) \\
&\qquad\qquad\qquad \Rightarrow \mathit{wp}(\langle E_{enter} \rangle, g_{on})
\end{aligned}
$$

Any refusal after performing E_{enter} must be a subset of $\{E_{enter}, E_{leave}, E_{off}\}$. In this case $\Sigma - X$ will always contain $\{E_{on}\}$ and thus will not block the guard g_{on} which is true after performing the *OPERATION* corresponding to the event E_{enter}.

This is enough to establish the following lemma. It states that the specification of being able to ensure the guard of a possible next event is true for all traces of a body of a process and is preserved by recursive calls. This lemma is in the context of divergence freedom, since in its proof we refer to maximal terminating traces of a process and need to make sure that the *CLI* can be established at the end of the sequence of *OPERATIONS* corresponding to those events. In more detail, we look at an arbitrary process but divergence freedom must be true for any process within the family of processes. This is why we state that *G preserves CLI* in the lemma.

Lemma 2. *If $\underline{Y} = G(\underline{Y})$ is a mutual recursive process such that G preserves CLI, meets Condition 1 then*

$$\forall p \bullet \underline{Y}_p \ \mathbf{sat}\ \mathit{ge}(tr, X) \Rightarrow \forall p \bullet \underline{G}(\underline{Y})_p \ \mathbf{sat}\ \mathit{ge}(tr, X)$$

where $\mathit{ge}(tr, X) = \mathit{CLI} \wedge I \wedge c_b = p \Rightarrow \mathit{wp}(tr, \bigvee_{x \in \Sigma - X} g_x)$

Now we can state the following theorem. If the guards of at least one of the *OPERATIONS* corresponding to the events offered for execution are enabled then not all the events combined with their *OPERATIONS* can be refused. The failures in the theorem are those given by the failures divergences model. However, we can move freely between the two semantic models since *CLI* guarantees that $(\mathit{LOOP} \mid\mid M)$ is divergence-free so the failures from the CSP will be the same in both models as stated in Section 2.2.

```
INITIALISATION
nn := 0
OPERATIONS
up ≙ SELECT nn = 0 ∨ nn = 1
    THEN nn := 1 END
down ≙ SELECT nn = 1
    THEN nn := 0 END
END
```

$$\begin{aligned} S(0) &= E_{up} \rightarrow S(1) \\ S(1) &= (E_{up} \rightarrow S(1)) \\ &\quad \Box\ (E_{down} \rightarrow block \rightarrow STOP) \end{aligned}$$

Fig. 3. Example terminating control executive and *MACHINE*

Theorem 1. *If LOOP* **sat** $ge(tr, X)$ *then* $\forall\, tr \bullet (tr, \Sigma) \notin failures(LOOP \parallel M)$

The corollary follows immediately.

Corollary 1. *If Condition 1 holds for the body of LOOP, then* $(LOOP \parallel M)$ *is deadlock-free.*

4 A coupling for terminating loops

In this section we augment the control language to include atomic events to model terminating loops. We then discuss the impact of modelling such loops on our notion of deadlock. We also modify the proof condition to accommodate this change and verify consistency of these new loops.

4.1 Extended syntax

The new syntax is defined as follows:

$$R ::= a \rightarrow R \mid R_1 \Box R_2 \mid S(p) \mid block \rightarrow STOP$$

STOP can be used to terminate a process, however it has no traces. The way in which we built the combined view above was to examine sequences of events and their corresponding sequences of *OPERATIONS*. We cannot simply use *STOP* because we need an event which appears in a CSP trace which we will map to an *OPERATION* in the B. By using the special event named *block* we have an event which can appear explicitly in a CSP trace and which corresponds to the guarded substitution, *SELECT false THEN skip END*. The reason for the guard of block, g_{block}, being false will be explained in the following section.

An example of a *MACHINE* and its control executive which includes *block* is shown in Figure 3.

4.2 Acceptable deadlock

The new syntax we introduced above means that deadlocks can be explicitly introduced into the CSP by the event prefix $block \rightarrow STOP$. These explicit deadlocks in the CSP are acceptable since we take the appearance of *block* in a control executive to indicate that termination is acceptable at that point. However, we do not wish to allow unexpected deadlocks which are introduced via the B as we discussed earlier. Therefore, in our correspondence between CSP and B, the notion of deadlock freedom we require is that not all of the *OPERATIONS* offered in the CSP are actually blocked (with false guards) in the B. However, we will not worry about deadlock if *block* can be the next event of a trace since that indicates acceptable deadlock at that point (e.g. after a controlled shutdown).

4.3 Modified Condition for deadlock freedom

The following condition is very similar to Condition 1. The only difference is that we restrict the traces that need to be examined.

Condition 2 $\forall\, tr \bullet block \notin tr \wedge$

$$tr \frown \langle block \rangle \notin traces(R_p[\underline{DIV} \,/\, \underline{S}]) \wedge$$
$$tr \in traces(R_p[\underline{DIV} \,/\, \underline{S}]) \wedge$$
$$CLI \wedge I \wedge c_b = p \Rightarrow wp(tr, PAIRS(tr, R_p))$$

This condition states that for all traces, tr, of the body of R_p which do not lead to blocking and given the invariants hold before the body is executed with the appropriate value of the control variables, the state reached after that trace enables a guard of at least one of the *OPERATIONS* corresponding to the next possible events.

For example, for the process $S(1)$ in Figure 3, we only need to check the following traces, $\langle\rangle$ and $\langle E_{up} \rangle$. We do not need to check $\langle E_{down} \rangle$ and $\langle E_{down}, block \rangle$ since they do not satisfy Condition 2. Given this restriction on the traces that need to be checked the definition of PAIRS remains unaffected, i.e. we do not need to provide a definition for the $block \rightarrow Stop$ case since it will never be needed. We do not check the traces which lead to blocking since deadlock is explicitly permitted in such cases.

Therefore, given the process $E_c \rightarrow block \rightarrow Stop$ only $CLI \wedge I \Rightarrow wp(\langle\rangle, g_c)$ needs to be checked. The system should not deadlock before the event E_c occurs.

4.4 Verifying deadlock freedom consistency for terminating loops

All the lemmas and theorems introduced so far are based on processes which do not include the *block* event. Now consider processes which may contain the *block* event. We will obtain a similar result to Lemma 2 which takes the *block* event into account. Consider again the process $S(1)$ from Figure 3. Its stable failures on the empty trace and the singleton trace $\langle E_{down} \rangle$ are

$$\{(\langle\rangle, X) \mid X \subseteq \{block\}\}$$
$$\{(\langle E_{down} \rangle, X) \mid X \subseteq \{E_{up}, E_{down}\}\}$$

Note that in the initial case the maximal X (ignoring *block*) is $\{\}$, and so $\Sigma - X = \{E_{up}, E_{down}\}$ and so $CLI \wedge I \wedge c_b = 1 \Rightarrow wp(\langle\rangle, g_{up} \vee g_{down})$ must hold in order to satisfy the specification in lemma 2. Following $\langle E_{down} \rangle$ we would naïvely need to show that $CLI \wedge I \wedge c_b = 1 \Rightarrow wp(\langle E_{down} \rangle, g_{block}) = wp(\langle E_{down} \rangle, false)$ which does not always hold. However, in this case we do not need to concern ourselves with satisfying the deadlock freedom specification since deadlock has been explicitly permitted by the inclusion of the *block* event in the process description. The above failures provide an insight on the extra predicate that needs to be added to the specification so that we focus only on establishing deadlock freedom for the appropriate traces. For a given trace, if we need to ensure that a next possible guard is enabled, *block* will be able to augment the refusal set since it should not be possible in the trace. On the other hand if explicit blocking occurs next then *block* is not in the refusals. Thus, the following lemma states that all traces which do not lead to explicit blocking are deadlock-free. This gives another property which is preserved by recursion.

Lemma 3. *If $\underline{Y} = G(\underline{Y})$ is a mutual recursive process such that G preserves CLI, meets Condition 2 and which can contain the block event then*

$$\forall\, p \bullet \underline{Y}_p \ \mathbf{sat}\ new\_ge(tr, X) \Rightarrow \forall\, p \bullet \underline{G(\underline{Y})}_p \ \mathbf{sat}\ new\_ge(tr, X)$$

where $new\_ge(tr, X) = CLI \wedge I \wedge c_b = \rho[p] \wedge block \in X \Rightarrow wp(tr, \bigvee_{x \in \Sigma - X} g_x)$

Now we can state the theorem that if the trace cannot be extended by *block* then not all the events combined with their *OPERATIONS* can be refused. Thus all deadlocks are marked by *block* in the CSP. The theorem relies upon the stable failures axiom in CSP which states that given a trace and a refusal any event can be either appended to the trace or added to the refusals. It also relies on the property of subset closure in the refusals of a behaviour.

Theorem 2. *If $LOOP$ **sat** $new\_ge(tr, X)$ then*
$\forall\, tr \bullet tr \frown \langle block \rangle \notin traces(LOOP) \Rightarrow (tr, \Sigma) \notin failures(LOOP \parallel M)$

The corollary follows immediately.

Corollary 2. *If Condition 2 holds for the body of LOOP, then $(LOOP \parallel M)$ is deadlock-free.*

5 Allowing channels in loops

This section follows the same pattern as the previous sections. We first extend the control language to include structured events and boolean expressions. We then describe what effect these new events have on deadlock freedom.

5.1 Further extended control syntax

The new syntax is given as follows;

$$\begin{aligned} R ::= {} & a \rightarrow R \mid R_1 \,\Box\, R_2 \mid \mathit{if}\ x\ \mathit{then}\ R_1\ \mathit{else}\ R_2\ \mathit{end}\ \mid c?x : T \rightarrow R \mid \\ & S(p) \mid \mathit{block} \rightarrow \mathit{STOP} \end{aligned}$$

The conditional term is defined as in Section 2.1. The input term behaves as described in Section 2.1.

In [15] we discussed the flow of information from the CSP into the B description. In particular, we distinguished between the environments of a control executive. We discussed the existence of an environment for the whole system which is external to both the CSP and B descriptions. The input term models an input from this environment which is then passed into the B specifications. We have only considered one input, we could easily extend the approach to deal with many inputs. We also described that all the outputs from the CSP originated in the B. In our restricted language there are no terms with the standard CSP syntax for output over a channel ($c!v \rightarrow P$). In our combination this would correspond to both the control executive and the B description setting the output value. The CSP is simply driving the *OPERATIONS* and has no part in constraining the values of the outputs. Instead, we introduced a new piece of syntax $c \,\textbf{?}\, v \rightarrow P$ to have precisely the CSP output semantics. The difference is that we view this term as the control executive passing information into the B specification. Further discussion of the use of outputs channels can be found in the technical report [16].

However, for a control executive to be well formed all of its variables must be bound. Variables are bound either by inputs from the external environment or from the B description or by appearing as parameters of the mutual recursion. An example of a control executive and its associated Abstract System based on this new syntax is given in Figure 4. This example meets Condition 2 and is deadlock-free. It illustrates how the variables x and f are bound by the environment and the parameter of the process L respectively. The functionality of the example is the servicing of lift requests. The lift is at a particular floor f and the control executive accepts requests to move to floor x. It proceeds to ascend or descend to the requested floor as appropriate.

5.2 Preserving consistency with new syntax

In Section 4 we considered loops that could terminate and ignored traces which contained the *block* event in Condition 2. With the new syntax above we do need to consider the traces of the bodies of the processes of a control executive which include an input over a channel and those which have been influenced by the branching of the boolean condition. Therefore, we need to change the definition of *PAIRS* to include cases for inputs and conditional expressions as given in Figure 5. For the term which inputs a value from the environment, if the trace is empty a guard is present so that for some CSP inputs the corresponding guard

```
INITIALISATION
floor := 0 || req := 0
OPERATIONS
request(xx) ≙ PRE xx : NAT THEN
          SELECT xx ≠ floor
          THEN req := xx END
          END
ascend ≙ SELECT req > floor
         THEN floor := req END
descend ≙ SELECT req < floor
         THEN floor := req END
END
```

$$\begin{aligned} LIFT &= L(0) \\ L(f) &= E_{request}?x : \mathbb{N} \rightarrow \\ &\quad if\ (x > f)\ then\ E_{ascend} \rightarrow L(x) \\ &\qquad\qquad else\ E_{descend} \rightarrow L(x) \end{aligned}$$

Fig. 4. Lift control executive and its associated *MACHINE*

$$\begin{aligned} PAIRS(\langle\rangle, E_a?x_c : T \rightarrow R)_\rho &= \exists\, x_b \in T \bullet g_a(x_b)) \\ PAIRS(\langle E_a.v\rangle \frown tr, E_a?x_c : T \rightarrow R)_\rho &= PAIRS(tr, R(v))_\rho \quad \text{where } v : T \\ PAIRS(tr, if\ x_c\ R_1\ else\ R_2\ end)_\rho &= \rho[x_c] \Rightarrow PAIRS(tr, R_1)_\rho \\ &\quad \wedge \neg\rho[x_c] \Rightarrow PAIRS(tr, R_2)_\rho \end{aligned}$$

Fig. 5. *PAIRS* definition

in the B is true. For example, in Figure 4 the *OPERATION request* restricts its function to inputs which are not of the same floor as the current floor of the lift. Execution is prohibited when the input is to the same floor as it is on. Therefore, in general blocking may occur on some inputs and the guard need not be true for all inputs, but deadlock will not occur provided at least one input is not blocked.

In the *PAIRS* clauses in Figure 5 we introduced a binding which is used to track updates to CSP variables. Use of the binding, ρ, to keep track of the values of variables is a standard technique in denotational semantics [12, 14]. In the clause containing the conditional term we use it to extract the value of x_c so that the appropriate predicate related to the guards of either R_1 or R_2 is offered.

Note also that the control predicate in Condition 2 can no longer simply refer to the value of the index in the parameter of the process. We will need the predicate $c_b = \rho[p]$. The value of c_b equals the value of the expression and will be contained in ρ. It will be used to track the values associated with the variables in the CSP description. For example, the process L in Figure 4 gives rise to the control predicate $c_b = \rho[f]$ when considering traces of its process body.

```
**MACHINE** $embedded\_switch$
**VARIABLES** $person$ , $status$
**INVARIANT** $person \in \mathbb{N} \wedge status \in 0 \,..\, 1 \wedge$
$person \leq 5$
**INITIALISATION** $person := 0 \;\|\; status := 0$

**OPERATIONS**
  $on \mathrel{\widehat{=}}$ **PRE** $person > 0 \wedge status = 0$ **THEN**
     $status := 1$
   **END** ;
  $off \mathrel{\widehat{=}}$ **PRE** $person = 1 \wedge status = 1$ **THEN**
     $status := 0$
   **END** ;
  $enter \mathrel{\widehat{=}}$ **SELECT** $person < 5$ **THEN**
     $person := person + 1$
   **END** ;
  $leave \mathrel{\widehat{=}}$ **SELECT** $person > 0$ **THEN**
     $person := person - 1$
   **END**
**END**
```

Fig. 6. Revised Embedded light switch

Thus from the above, the impact of the new syntax on Condition 2 is minimal. We simply need to take the binding ρ in account and provide additional clauses for *PAIRS*. Their impact on the verification of consistency is also minimal. The theorems and corollaries remain unchanged. In the supporting lemmas the binding has to be taken into consideration in order to be able to consider bounded expressions. Therefore, when we examine the set of failures of a process we need to look at the set of failures under ρ, i.e. $failures[\![P]\!]_{\rho}$. We also have to add new cases in the structural induction of Lemma 1 for input and conditional terms. These changes are technical details. What we have to prove and the structure of the proof to demonstrate that we are deadlock-free remain the same.

6 Example with divergence and deadlock freedom

We conclude this paper by showing how the divergence and deadlock freedom verification can be applied to a small example in our style of specification. Earlier in Figure 1 we considered an embedded light switch consisting of guarded statements. We can also interpret this example with the light switch as the software to be developed and the room as its where people can enter and leave. Therefore, the *OPERATIONS on* and *off* will be defined by pre-conditioned statements and not guarded ones as before and so we have the *MACHINE* shown in Figure 6.

The two conditions for divergence freedom for this system were outlined in Section 3.3 and are discussed in more detail in [15]. They state that the initialisation of the *MACHINE* must establish the control loop invariant, *CLI*, and that any sequence of *OPERATIONS* corresponding to a trace of events of the body of the process *ROOM* must preserve the *CLI*.

$$[INITIALISATION]\,CLI$$
$$CLI \wedge I \wedge c_b = 0 \Rightarrow [enter;\ on;\ off;\ leave]\,CLI$$
$$\text{where } CLI = person = 0 \wedge status = 0 \text{ and}$$
$$I = person \in \mathbb{N} \wedge status \in 0..1 \wedge person \leq 5$$

In Section 3.5 we stated the five conditions to be verified to ensure deadlock freedom. The guards of the *OPERATIONS* *on* and *off* are merely true and so the second and third proof obligations are simplified to show termination.

$$CLI \wedge I \wedge c_b = 0 \Rightarrow wp(\langle E_{enter} \rangle, true)$$
$$CLI \wedge I \wedge c_b = 0 \Rightarrow wp(\langle E_{enter}, E_{on} \rangle, true)$$

Since the *CLI* can be established for the above sequence of *OPERATIONS* (*enter*; *on*; *off*; *leave*) then we can infer that any prefix of this sequence terminates, and hence establishes true. Therefore, we do not have to check the above proof obligations nor the fifth one. Thus we are reduced to verifying the remaining proof obligations:

$$CLI \wedge I \wedge c_b = 0 \Rightarrow person < 5$$
$$\text{and } CLI \wedge I \wedge c_b = 0 \Rightarrow wp(\langle E_{enter}, E_{on}, E_{off} \rangle, person > 0)$$

and these are both easily established.

If we had failed to establish that *ROOM* preserves *CLI* then it might just be that the *CLI* is not appropriate. Alternatively, it could be that one of the *OPERATIONS* *on* or *off*, from Figure 6 were called outside their pre-conditions in which case there is a divergence and hence no possible *CLI*. For example, the sequence (*enter*; *leave*; *on*; *off*) fails to establish the *CLI*. Even if we can establish the *CLI* we must still check for deadlocks since *CLI* may have been established miraculously. For example, the sequence (*enter*; *leave*; *leave*) can establish the *CLI* since the guard of the second *leave* is false and so anything could be true of the final state including the *CLI*. However, $CLI \wedge I \wedge c_b = 0 \Rightarrow wp(\langle E_{enter}, E_{leave} \rangle, g_{leave})$ does not hold, and so we discover a potential deadlock. Thus the control executive $S(0) = E_{enter} \rightarrow E_{leave} \rightarrow E_{leave} \rightarrow S(0)$ is unsuitable. Control executives are only suitable if they are divergence- and deadlock-free. Given a suitable executive we can then specify safety and liveness properties in terms of events. For example, we could say that the light switch will be enabled when a person enters the room.

7 Discussion

Other work in combining a process algebra and a state based-method has predominantly centered around combinations of CSP or Timed-CSP together with

Z or Object Z [6]. Some of the approaches introduce new semantics whereas our semantic combination preserves consistency of the original semantics of both languages so that each description could be analysed separately (potentially with the powerful tool support currently available), but links are drawn between them so that the events in a process can be interpreted from a different viewpoint. The CSP description cannot contribute to the computation of the individual state transitions. Many of these combinations split the input and output from the specification of the individual state transition. For example, Roscoe *et al.* [10] divide every schema into two events, one with input parameters and one for output. Our style is akin with Fischer [7] where each event maps to one *OPERATION*.

In the remainder of this section we briefly discuss our approach in relation to Butler's work which combines CSP and B [4]. He takes a CSP-like process and translates it into a B specification. Therefore in his approach the CSP could be simply a process which is translated into an event-based view of a system or it can be used to constrain the execution of a B *MACHINE* using the CONJOINS mechanism. The latter way of using CSP to drive a B *MACHINE* is similar to ours where we think of executing the CSP in parallel with a B *MACHINE*. However, the main difference is that in Butler's approach the CSP-like process is combined into the B specification and thus does not retain the two views separately as a CSP process with corresponding *OPERATIONS*. Instead, Butler introduces a new *MACHINE* where new *OPERATIONS* are defined corresponding to the events of the CSP process. If the CSP is being used to constrain the B *MACHINE* then the *OPERATIONS* from that *MACHINE* are called within the body of these new *OPERATIONS*. The guards of these new *OPERATIONS* correspond to new state variables and are like place holders to record the point of execution of the process. However, the approach does not focus on consideration of the new guards discharging the pre-conditions of the *OPERATIONS* of the Abstract System nor ensuring that their guards are enabled.

Another difference between Butler's approach and ours is that we disallow direct visibility of AMN state variables in a CSP process. In Butler's work the B state is directly visible in the CSP description which is appropriate since the CSP will be translated into B. However, we keep our descriptions separately and so the only way the CSP knows about B state is via information passed as parameters. This means that all relevant state information appears in traces and can be subject of trace specifications. No B state is directly visible within the CSP control executive. We go further and distinguish between the environments of the CSP and the external system with respect to the B Abstract System.

The notion of consistency introduced in this paper between the two views could also apply to Butler's work, although the csp2B translation does not itself ensure that the pre-conditions or the guards of the B OPERATIONS will be met. It may be the case that the notion of deadlock freedom for Butler's work need not be as complicated as ours and would simply need to show that one of the guards of the new combined specification was enabled in any legitimate state. This is likely to require a strengthening of the *MACHINE INVARIANT*. Such a deadlock-free condition for event-based AMN systems has already been pointed out by Abrial in [2]. This can also be done directly in our approach by using a control loop $S(0) = \Box\, E_{op} \rightarrow S(0)$ denoting an external choice over all the *OPERATIONS* in the loop. The only proof obligation we obtain from Condition 2 concerns the empty trace and reduces to $CLI \wedge I \wedge c_b = 0 \Rightarrow \bigvee g_{op}$. The *CLI* would have to be strong enough to imply the guard of one of the *OPERATIONS*. This *CLI* would correspond to the strengthening of the *MACHINE INVARIANT* likely to be required by Butler's approach. The above loop $S(0)$ is just a special case of our approach.

Finally, the focus of our work was to develop CSP control executives for sequential processors and define a style of specification which will allow some of the *OPERATIONS* to be refined to code. Extending the approach to include abstraction is a topic of current research.

Acknowledgements Thanks to the anonymous referees for their detailed and constructive comments.

References

1. Abrial J. R.: *The B Book: Assigning Programs to Meaning*, CUP (1996).
2. Abrial J. R.:*Extending B without Changing it (for Developing Distributed Systems).* In H. Habrias, editor, Proc. of the 1st B Conference, Nantes, France (1996).
3. Butler M.J.: *A CSP Approach to Action Systems.* D.Phil Thesis, Programming Research group, Oxford University (1992).
4. Butler M.J.:*csp2B: A Practical Approach to Combining CSP and B*, In J.M.Wing, J. Woodcock, J. Davies, editors, FM'99 World Congress, Springer (1999).
5. Dijkstra E. W.:*A Discipline of Programming*, Prentice-Hall (1976).
6. Fischer C.:*How to combine Z with a Process Algebra.* In J. Bowen, A. Fett, and M.Hinchey, editors, ZUM'98 The Z formal Specification Notation, volume 1493 of LNCS, Springer (1998).
7. Fischer C.: *CSP-OZ: A combination of Object-Z and CSP.* In H. Bowman and J. Derrick, editors, Formal Methods for Open Object-Based Distributed Systems (FMOODS'97), volume 2, Chapman & Hall (1997).
8. Hoare C.A.R.: *Communicating Sequential Processes*, Prentice Hall (1985).
9. Morgan C.C.:*Of wp and CSP.* In W.H.J. Feijen, A.J.M. van Gasteren, D. Gries and J. Misra, editors, *Beauty is our business: a birthday salute to Edsger W. Dijkstra.* Springer (1990).
10. Roscoe A. W., Woodcock J. C. P. and Wulf L.:*Non-interference through determinism.* In D. Gollmann, editor, ESORICS 94, volume 875 of LNCS, Springer (1994).
11. Roscoe A. W.:*The Theory and Practice of Concurrency*, Prentice-Hall, 1997.
12. Scattergood J.B.:*The Semantics of Machine-Readable CSP*, Oxford University D.Phil thesis, 1998.
13. Schneider S.:*Concurrent and Real-time Systems: The CSP approach*, Wiley, 2000.
14. Stoy, J. E.: *Denotational Semantics*, MIT Press (1977).
15. Treharne H., Schneider S.: *Using a Process Algebra to control B OPERATIONS.* In K. Araki, A. Galloway and K. Taguchi, editors, IFM'99, York, Springer (1999).
16. Treharne H., Schneider S.: *How to drive a B Machine (Full Version).* Technical Report CSD-TR-00-03, Royal Holloway, University of London, May (2000).