Verifying Controlled Components

Steve Schneider and Helen Treharne

Department of Computer Science, Royal Holloway, University of London

Abstract. Recent work on combining CSP and B has provided ways of describing systems comprised of components described in both B (to express requirements on state) and CSP (to express interactive and controller behaviour). This approach is driven by the desire to exploit existing tool support for both CSP and B, and by the need for compositional proof techniques. This paper is concerned with the theory underpinning the approach, and proves a number of results for the development and verification of systems described using a combination of CSP and B. In particular, new results are obtained for the use of the hiding operator, which is essential for abstraction. The paper provides theorems which enable results obtained (possibly with tools) on the CSP part of the description to be lifted to the combination. Also, a better understanding of the interaction between CSP controllers and B machines in terms of *non-discriminating* and *open* behaviour on channels is introduced, and applied to the deadlock-freedom theorem. The results are illustrated with a toy lift controller running example.

1 Introduction

Morgan's failures/divergences semantics for event systems [Mor90] enables the various CSP semantics to be given to B machines. These CSP semantics allow machines to be treated as CSP components within a concurrent system, and we can combine them with other CSP components using architectural operators such as parallel composition and abstraction.

Recent work [Tre00] has considered the interaction between a particular kind of B machine and a controller written as a (recursive) sequential CSP process. An important property of a controller for a machine is that it should invoke machine operations only within their preconditions. Previous results [Tre00] have identified conditions sufficient to guarantee $P \parallel M$ to be divergence-free for a controller P and machine M, which ensures this important property. These results require identification of a *control loop invariant* (CLI) on the state of the B machine M, which must be true on every recursive call. This is established by considering the semantics of the B operations as they are called within the controller, and essentially computing the weakest precondition required to establish the CLI.

In combining communicating B machines, we use a particular architecture [ST02b] to restrict the interaction between components, by ensuring that each B machine interacts only with its own controller. A system will be structured as

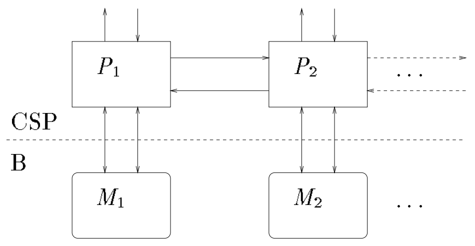

Fig. 1. A CSP and B combined system architecture

a collection of B machines $M_1, ..., M_n$, each with its own CSP controller process $P_1, ..., P_n$. A *controlled component* is the parallel combination of a controller and its B machine, of the form $P \parallel M$.

Each M_i is under the control of the corresponding P_i, and the P_i's can also interact with each other. This architecture is illustrated in Figure 1. Interaction across the system can occur only between the CSP processes. This approach enables compositional verification, whereby we are able to verify properties of the entire system by obtaining results about smaller structures within the system. In particular, both CSP and B already have mature tool support, which we aim to apply in the verification of combined systems.

The model-checker FDR [For97] performs model-checking on systems described in CSP, and is therefore suitable for analysing the controllers, individually and in combination. The paper provides theorems which enable results obtained (possibly with tools) on the CSP part of the description to be lifted to the combination[1]. We obtain a number of theorems in the various CSP semantic models.

In practice, we find that it is often the case that a property holds in a combined system for reasons associated with the state within the B components. In this case, the CSP controller descriptions need to be augmented with the relevant state information. This paper also provides theorems which support the required manipulations of CSP controllers. A fuller version of this paper [ST02a] gives rigorous proofs of all the theorems and lemmas. In this paper, we provide informal explanations of the theorems.

2 Background

2.1 CSP Events

CSP processes are defined in terms of the *events* that they can and cannot do. Processes interact by synchronising on events, and the occurrence of events is atomic. The set of all events is denoted by Σ.

[1] The FDR checks discussed in this paper are available at `http://www.cs.rhul.ac.uk/research/formal/steve/code/lifts.fdr2`

Events may be compound in structure, consisting of a *channel name* and some (possibly none) *data values*. Thus, events have the form $c.v_1...v_n$, where c is the channel name associated with the event, and the v_i are data values. The *type* of the channel c is the set of values that can be associated with c to produce events.

For example, if *trans* is a channel name, and $\mathbb{N} \times \mathbb{Z}$ is its type, then events associated with *trans* will be of the form $trans.n.z$, where $n \in \mathbb{N}$ and $z \in \mathbb{Z}$. For example, $trans.3.8$ is one such event.

A *partial event*, or (following [Sca98]) *partially completed datatype value* is a channel name together with some values, but not necessarily all. For example, $trans.3$ is a partial event. Any channel is a special case of a partial event.

Given a set of partial events PE, we can define the set of events $\{| \ PE \ |\}$ which are the completions of events in PE, as follows:

$$\{| \ PE \ |\} = \{p.w \mid p \in PE \wedge p.w \in \Sigma\}$$

We use alphabetised CSP, so every process has an alphabet, which is the set of events whose occurrence requires its participation. The alphabet of a process P is denoted $\alpha(P)$. For the purposes of this paper we will require that the alphabet of any process is given by a set of channels C, so that $\alpha(P) = \{| \ C \ |\}$.

2.2 CSP controllers

A controller for a B machine is a particular kind of CSP process. To interact with the B machine, it makes use of control channels which have both input and output, and provide the means for controllers to synchronise with B machines. For each operation $w \longleftarrow e(v)$ of a controlled machine with v of type $T_{in}(e)$ and w of type $T_{out}(e)$ there will be a channel e of type $T_{in}(e) \times T_{out}(e)$, so communications on e are of the form $e.v.w$.

Controller descriptions may also include assertions about the values of variables they are using. These are incorporated in CSP either as *blocking assertions* (which block if the assertion is false) or as *diverging assertions* (which diverge if the assertion is false), depending on the role they play in verification.

When we talk about a CSP controller P we mean a process which has a given set of control channels C. The controlled B machine will have exactly $\{| \ C \ |\}$ as its alphabet: it can communicate only on channels in C.

Controller syntax Controllers are generated from the following subset of the CSP syntax, as discussed in [ST02b].

$$\begin{aligned} P ::= {} & a \to P | c?x \to P | d!v \to P | e!v?x\{E(x)\} \to P | e!v?x\langle E(x)\rangle \to P | \\ & P_1 \ \Box \ P_2 | P_1 \sqcap P_2 | \bigsqcap_{x|E(x)} P | \ \mathsf{if}\ b\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 | S(p) \end{aligned}$$

where a and is a *synchronisation event*, c is a *communication channel* accepting inputs, d is a communication channel sending output values, e is a *control channel*, x represents all data variables on a channel, v represents all data values being passed along a channel, $E(x)$ is a predicate on x (it may be elided, in which case it is considered to be *true*), b is a boolean expression, and $S(p)$ is a process expression.

The process $a \rightarrow P$ is initially prepared to engage in an a event, after which it behaves as P. The input $c?x \rightarrow P$ is prepared to accept any value x along channel c, and then behave as P (whose behaviour can be dependent on x). The output $d!v \rightarrow P$ provides v as output. The operation call $e!v?x\{E(x)\} \rightarrow P$ is an interaction with an underlying B machine: the value v is passed from the process as input to the B operation, and the value x is accepted as output from the B operation. If x meets the condition $E(x)$ then the process behaves as P. If x does not meet the condition then the process diverges. On the other hand, $e!v?x\langle E(x)\rangle \rightarrow P$ only allows $e.v.x$ if $E(x)$, otherwise the event is blocked. Behaviour subsequent to $e.v.x$ is that of P.

The external choice process $P_1 \Box P_2$ is initially prepared to behave either as P_1 or as P_2, and the choice is resolved on occurrence of the first event. Binary and general internal choice are possible, though not used in the example presented here. The conditional choice $\mathsf{if}\ b\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2$ behaves as P_1 or P_2 depending on the evaluation of the condition b. Finally, the process expression $S(p)$ expresses a recursive call.

2.3 CSP semantic models

There are three semantic models used in this paper: the *Traces* model, the *Stable Failures* model, and the *Failures/Divergences* model. We introduce the relevant features of them here. Full details of these models can be found in [Ros97,Sch99].

Traces A *trace* is a finite sequence of events. A sequence tr is a trace of a process P if there is some execution of P in which exactly that sequence of events is performed. The set $traces(P)$ is the set of all possible traces of process P. The traces model for CSP associates a set of traces with every CSP process. If $traces(P) = traces(Q)$ then P and Q are equivalent in the traces model, and we write $P =_T Q$.

Stable Failures A *stable failure* is a pair (tr, X) consisting of a trace tr and a set of events X. Such a pair is a stable failure of a process P if there is some execution of P on which tr is the sequence of events performed, reaching a state in which all events in X can be refused, and also no internal progress is possible. The set $\mathcal{SF}\llbracket P \rrbracket$ is the set of stable failures of P. The stable failures model for CSP associates a set of stable failures, and a set of traces, with every CSP process. If $\mathcal{SF}\llbracket P \rrbracket = \mathcal{SF}\llbracket Q \rrbracket$ and also $traces(P) = traces(Q)$ then P and Q are equivalent in the stable failures model and we write $P =_{SF} Q$.

Failures and Divergences A *divergence* is a finite sequence of events tr. Such a sequence is a *divergence* of a process P if it is possible for P to perform an infinite sequence of internal events (such as a livelock loop) on some prefix of tr. The set of divergences of a process P is written $\mathcal{D}\llbracket P \rrbracket$.

A *failure* is a pair (tr, X) consisting of a trace tr and a set of events X. It is a failure of a process P if either tr is a divergence of P (in which case X can be any set), or (tr, X) is a stable failure of P. The set of all possible failures of a process P is written $\mathcal{F}[\![P]\!]$. If $\mathcal{D}[\![P]\!] = \mathcal{D}[\![Q]\!]$ and $\mathcal{F}[\![P]\!] = \mathcal{F}[\![Q]\!]$ then P and Q are equivalent in the failures/divergences model, written $P =_{FD} Q$.

The different models are used to analyse CSP systems with respect to different properties. This paper is concerned with the failures-divergences model is used to check for liveness properties such as divergence-freedom. If a system description includes the possibility of divergence (for example, if it includes internal events), then it is necessary to use the failures divergences model to check for divergence-freedom.

An important relationship between the stable failures model and the failures divergences model is that if a process is divergence-free (i.e. its set of divergences is empty), then its failures are the same as its stable failures. This is captured in the following theorem:

Theorem 1. *If* $\mathcal{D}[\![P]\!] = \{\}$, *then* $\mathcal{F}[\![P]\!] = \mathcal{SF}[\![P]\!]$.

This theorem is useful because it allows us to carry out analysis in the stable failures model, which is more efficient, and to establish results which remain valid in the failures divergences model. For example, if a process P is divergence-free, then to check that it is deadlock-free (i.e. that $(tr, \alpha(P))$ cannot be a failure of P for any tr), it is sufficient to check this in the stable failures model (that $(tr, \alpha(P))$ cannot be a stable failure). The model-checker FDR [For97] can carry out divergence-freedom and deadlock-freedom checks mechanically.

2.4 CSP semantics for B machines

Morgan's CSP-style semantics [Mor90] for event systems enables us to define such semantics for B machines. A machine M thus has a set of traces $\mathcal{T}[\![M]\!]$, a set of failures $\mathcal{F}[\![M]\!]$, and a set of divergences $\mathcal{D}[\![M]\!]$. A sequence of operations $\langle e_1, e_2 \ldots e_n\rangle$ is a *trace* of M if it can possibly occur. This is true precisely when it is not guaranteed to be blocked, or in other words it is not guaranteed to achieve *false*. In *wp* notation we write $\neg wp(e_1;\ e_2;\ \ldots;\ e_n, \mathit{false})$, or in Abstract Machine Notation $\neg([e_1;\ e2;\ \ldots;\ e_n]\mathit{false})$. (The empty trace is treated as *skip*). A sequence does not diverge if it is guaranteed to terminate (i.e. establish *true*). Thus, a sequence is a divergence if it is not guaranteed to establish *true*, i.e. $\neg([e_1;\ e2;\ \ldots;\ e_n]\mathit{true})$. Finally, given a set of events X, each event $e \in X$ is associated with a guard g_e. A sequence with a set of events is a *failure* of M if the sequence is not guaranteed to establish the disjunction of the guards. Thus, $(e_1;\ e2;\ \ldots;\ e_n, X)$ is a failure of M if $\neg[e_1;\ e2;\ \ldots;\ e_n](\bigvee_{e \in X} g_e)$. More details of the semantics of B machines can be found in [Tre00]

Morgan does not give a stable failures semantics for action systems. We will define the stable failures $\mathcal{SF}[\![M]\!]$ for a machine M in terms of its failures divergences semantics, as follows:

Definition 1. *The stable failures of a B machine are defined as follows:*

$$\mathcal{SF}[\![M]\!] = \{(tr, X) \mid (tr, X) \in \mathcal{F}[\![M]\!] \wedge tr \notin \mathcal{D}[\![M]\!]\}$$

```
MACHINE     i_Lift
VARIABLES   i_floor
INVARIANT   i_floor : NAT
INITIALISATION  i_floor := 0
OPERATIONS

  i_inc(nn) =
     PRE nn : NAT1
     THEN i_floor := i_floor + nn
     END;

  i_dec =
     PRE i_floor > 0
     THEN i_floor := i_floor - 1
     END;

bb <-- i_isZero =
     IF i_floor = 0
     THEN bb := TRUE
     ELSE bb := FALSE
     END
END
```

```
$i\_LiftCtrl =$
  $i\_up?y \rightarrow i\_inc!y \rightarrow i\_LiftCtrl$
  $\Box\ i\_down?y \rightarrow i\_DOWN(y)$
  $\Box\ i\_ground \rightarrow i\_LOWER$

$i\_DOWN(n) =$
  if $n = 0$
  then $i\_LiftCtrl$
  else $i\_isZero?bb \rightarrow$
       if $(bb = TRUE)$
       then $i\_LiftCtrl$
       else $i\_dec \rightarrow i\_DOWN(n-1)$

$i\_LOWER =$
  $i\_isZero?bb \rightarrow$
       if $(bb = TRUE)$
       then $i\_LiftCtrl$
       else $i\_dec \rightarrow i\_LOWER$
```

Fig. 2. A Lift machine `i_Lift` and its controller $i\_LiftCtrl$

Observe that with this definition, Theorem 1 also holds for B machines M.

We have a technique [Tre00,ST02b], based on control loop invariants, for establishing that a combination $P \parallel M$ is divergence-free. In other words, previous results provide a means to establish that $\mathcal{D}[\![P \parallel M]\!] = \{\}$. This paper is not concerned with that technique. Rather we are concerned with composing together a number of $P_i \parallel M_i$ pairs once we have established that $\mathcal{D}[\![P_i \parallel M_i]\!] = \{\}$ for each pair. Hence a number of the theorems in this paper will include an assumption that $\mathcal{D}[\![P_i \parallel M_i]\!] = \{\}$. The assumption in particular cases can be discharged using the control loop invariant technique.

3 A motivating toy example: a lift controller

As motivation for the results presented in this paper, we consider a toy example of a collection of lift machines described in B, controlled by CSP controller processes. An individual lift is given in Figure 2. It describes a particular lift, indexed by i. We will then go on to define a system consisting of a collection of such lifts.

3.1 Individual lifts

The Lift machine provides three operations: `i_inc(nn)` which moves the lift up nn floors, `i_dec` which moves the lift down one floor, and a query operation `i_isZero` which indicates whether or not the lift is on the ground floor.

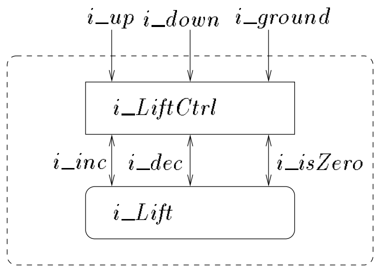

Fig. 3. The controlled lift system

The CSP controller is also given in Figure 2. It interacts with a user through the events *i_up*, *i_down*, and *i_ground*, and controls the lift accordingly:

- on *i_up.y*, it calls `i_inc` and moves the lift up y floors.
- on *i_down.y*, it calls `i_dec` y times or until it reaches the ground if this is sooner.
- on *i_ground*, it is required to move the lift to the ground floor. To do this, it repeatedly checks (using `i_isZero`) whether the lift is on the ground floor, and if not then it moves the lift down a floor with `i_dec`.

We are firstly interested in each controlled lift combination

$$i\_LiftSys = (i\_Lift \parallel i\_LiftCtrl) \setminus \{\mid i\_inc, i\_dec, i\_isZero \mid\}$$

which is pictured in Figure 3. We require as a minimum that this combination is deadlock-free and divergence-free.

These properties are apparent in this simple example. Deadlock-freedom is immediate because the B machine is always willing to engage in any event required by the controller, and the controller itself is either waiting for an interaction from its environment or else ready to call a controller operation. Divergence could arise either (i) from a B operation being called outside its precondition, or (ii) from an infinite sequence of internal events. In the case of (i), the only operation with a non-trivial precondition is `i_dec`, and the controller is constructed so that `i_dec` is only ever called when the lift is not at floor 0. In the case of (ii), the lift will eventually reach the ground floor and so an infinite sequence of calls of `i_dec` cannot occur.

In more complex examples the properties may not be so apparent, and it would be useful to be able to apply analysis tools to carry out model-checking on the combined system. However, no tools currently exist which can analyse a combination of B and CSP descriptions. The best we can aim for is to analyse the descriptions separately and combine results. In particular, for considering properties such as deadlock and livelock we would aim to apply a tool such as FDR [For97] to the CSP part of the description, and deduce results about the controlled combination. In particular, once it has been established that the

```
$i\_LiftCtrl2(f) =$
  $i\_up?y \rightarrow i\_inc!y \rightarrow i\_LiftCtrl2(f + y)$
  $\Box\ i\_down?y \rightarrow i\_DOWN2(f, y)$
  $\Box\ i\_ground \rightarrow i\_LOWER2(f)$

$i\_LOWER2(f) =$
  $i\_isZero?bb \boxed{\{bb = TRUE \Leftrightarrow f = 0\}} \rightarrow$
      if $(bb = TRUE)$
      then $i\_LiftCtrl2(f)$
      else $i\_dec \rightarrow i\_LOWER2(f - 1)$

$i\_DOWN2(f, n) =$
  if $n = 0$
  then $i\_LiftCtrl2(f)$
  else $i\_isZero?bb$
      $\boxed{\{bb = TRUE \Leftrightarrow f = 0\}} \rightarrow$
    if $(bb = TRUE)$
    then $i\_LiftCtrl2(f)$
    else $i\_dec \rightarrow$
        $i\_DOWN2(f - 1, n - 1)$
```

Fig. 4. The controller with diverging assertions

controller does not call operations outside their precondition, then the aim is that all deadlocking and divergent behaviour is essentially contained in the controller and can be identified without further reference to the B machine.

It has previously been established [ST02b] that, under appropriate conditions, the deadlock-freedom of a controller P implies the deadlock-freedom of a controlled combination $P \parallel M$. This result appears in this paper as Theorem 2.

We also establish in this paper (Theorem 3) that, under appropriate conditions, if $P \setminus E$ is divergence-free, then so too is $(P \parallel M) \setminus E$.

These two theorems are exactly what is required. We have only to check that $i\_LiftCtrl$ is deadlock-free to deduce the same for $i\_LiftSys$. And we have only to check that $i\_LiftCtrl \setminus \{\mid i\_inc, i\_dec, i\_isZero \mid\}$ is divergence-free to deduce this for $i\_LiftSys$. These are both checks that are easily done using FDR.

However, the second check turns out not to be correct. The description of $i\_LiftCtrl \setminus \{\mid i\_inc, i\_dec, i\_isZero \mid\}$ in fact contains a divergence arising from the infinite sequence $\langle i\_ground, i\_isZero.false, i\_dec, i\_isZero.false, i\_dec, \ldots\rangle$ of $i\_LiftCtrl$. It is the machine i_Lift that ensures that this cannot occur — but that machine was not included in the FDR analysis.

The problem is that some of the control flow is dependent on the state information maintained in the B machine, and so the useful theorems we have available are not directly applicable. We need to include the relevant state information in the description of the CSP controller, and also the expectation that the value *true* will be received on channel i_isZero exactly when $f = 0$. This is included as an assertion, as shown in Figure 4. It is straightforward to show that $i\_LiftCtrl2(0)$ is an appropriate driver for i_Lift (using control loop invariant $f = floor$). The proof that $i\_LiftCtrl2(0) \parallel i\_Lift$ has no divergences involves establishing the truth of the assertion for the input bb on i_isZero.

Introducing a diverging assertion means that $i\_LiftCtrl2(0)$ trivially has a divergence (i.e. the behaviour when the assertion is not met), so it is not appropriate to check $i\_LiftCtrl2(0) \setminus \{\mid i\_inc, i\_dec, i\_isZero \mid\}$ for divergence-freedom. However, in the context of i_Lift we know the assertion will always be

```
$i\_LiftCtrl3(f) =$
  $i\_up?y \to i\_inc!y \to i\_LiftCtrl3(f+y)$
  $\Box\ i\_down?y \to i\_DOWN3(f,y)$
  $\Box\ i\_ground \to i\_LOWER3(f)$

$i\_LOWER3(f) =$
  $i\_isZero?bb \boxed{\langle bb = TRUE \Leftrightarrow f = 0 \rangle} \to$
      if $(bb = TRUE)$
      then $i\_LiftCtrl3(f)$
      else $i\_dec \to i\_LOWER3(f-1)$

$i\_DOWN3(f,n) =$
  if $n = 0$
  then $i\_LiftCtrl3(f)$
  else $i\_isZero?bb$
      $\boxed{\langle bb = TRUE \Leftrightarrow f = 0 \rangle} \to$
    if $(bb = TRUE)$
    then $i\_LiftCtrl3(f)$
    else $i\_dec \to$
        $i\_DOWN3(f-1, n-1)$
```

Fig. 5. The controller with blocking assertions

true, so we may replace the diverging assertion by a blocking one, and yield a controller with the same behaviour in the context of *i_Lift*. The only difference is that this controller blocks rather than diverges when the assertion is false, and since the assertion is never false in the context of *i_Lift*, the resulting behaviour is the same. This transformation is justified by Corollary 1. Thus, we obtain a variant $i\_LiftCtrl3(0)$ of the controller, given in Figure 5, such that $i\_LiftCtrl3(0) \parallel i\_Lift = i\_LiftCtrl2(0) \parallel i\_Lift$.

Now we have a transformation of the controller which is divergence-free when the internal events are hidden: $i\_LiftCtrl3(0) \setminus \{| \ i\_inc, i\_dec, i\_isZero \ |\}$ is divergence-free, and this can be checked using FDR (given a bound on the number of possible consecutive *i_up* events). So we can conclude that $(i\_LiftCtrl3(0) \parallel i\_Lift) \setminus \{| \ i\_inc, i\_dec, i\_isZero \ |\}$ is divergence-free.

Now Corollary 1 also allows the assertions of $i\_LiftCtrl2(0)$ to be dropped completely, resulting in a controller whose behaviour does not depend on the value of the parameter f at all, and which is therefore equivalent to *i_LiftCtrl*. This transformation is discussed in more detail in [ST02a]. We have therefore now established divergence-freedom of the original combination $(i\_LiftCtrl \parallel i\_Lift) \setminus \{| \ i\_inc, i\_dec, i\_isZero \ |\}$.

To sum up: we identified two new controllers which are equivalent in the presence of *i_Lift* to the original controller *i_LiftCtrl*, and which are each used in a different part of the proof.

$$i\_LiftCtrl2(0) \parallel i\_Lift = i\_LiftCtrl3(0) \parallel i\_Lift = i\_LiftCtrl \parallel i\_Lift$$

- The combination $i\_LiftCtrl2(0) \parallel i\_Lift$ can be shown to be divergence-free using techniques from [ST02b].
- $i\_LiftCtrl3(0) \setminus \{| \ i\_inc, i\_dec, i\_isZero \ |\}$ is divergence-free, and so $(i\_LiftCtrl3(0) \parallel i\_Lift) \setminus \{| \ i\_inc, i\_dec, i\_isZero \ |\}$ is divergence-free.
- And $i\_LiftCtrl2(0) \parallel i\_Lift$ is equivalent to the original $i\_LiftCtrl \parallel i\_Lift$.

These results together establish the required result: that the original combination $(i\_LiftCtrl \parallel i\_Lift) \setminus \{| \ i\_inc, i\_dec, i\_isZero \ |\}$ is divergence-free. The state

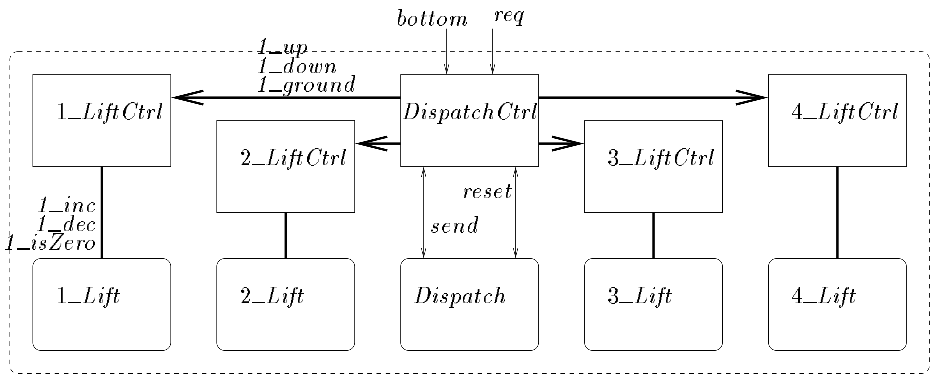

Fig. 6. The complete system *Lifts*

information was introduced into the controller purely to enable the verification to take place, and can be removed once the result has been established.

We also deduce that $(i\_LiftCtrl \parallel i\_Lift) \setminus \{|\ i\_inc, i\_dec, i\_isZero\ |\}$ is deadlock-free. This follows from deadlock-freedom of $i\_LiftCtrl \parallel i\_Lift$.

3.2 A collection of lifts

We will now combine the lifts into a single system together with a *Dispatch* and *DispatchCtrl* component which manages requests for lifts from buttons on the various floors. When a request for a lift is made from a particular floor, only one of the lifts needs to be sent. An example architecture made up of four lifts is pictured in Figure 6.

The *Dispatch* machine contains some algorithm for deciding which lift should be sent to a particular floor. It has an operation $ii, nn, dd \longleftarrow send(ff)$. On input of the floor *ff* to send a lift to, it provides as output the lift *ii* to be sent, the number of floors *nn* and the direction *dd* that lift *ii* will need to travel (as computed by *Dispatch*). *Dispatch* has another operation *reset*, which is called when all lifts return to the ground floor. The particular details of *Dispatch* are not relevant to this example and will not be given here.

The *DispatchCtrl* controller accepts requests along channel *req*: an input $req?x$ is a request for a lift to go to floor x. It makes use of the *Dispatch* machine to decide which lift to allocate, and then sends the appropriate instruction to the relevant lift. The controller can also accept an instruction *bottom* to return all lifts to the ground floor. It is defined as follows:

$$\begin{aligned}
DispatchCtrl = {} & req?x \rightarrow send!x?i?n?d \rightarrow \ \mathsf{if}\ d = ascend \\
& \qquad\qquad\qquad\qquad\qquad\quad \mathsf{then}\ i\_up!n \rightarrow DispatchCtrl \\
& \qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\ i\_down!n \rightarrow DispatchCtrl \\
& \square\ bottom \rightarrow 1\_ground \rightarrow 2\_ground \rightarrow 3\_ground \\
& \qquad\qquad \rightarrow 4\_ground \rightarrow reset \rightarrow DispatchCtrl
\end{aligned}$$

Our overall system is then composed of the controlled lift components $\mathit{Lifts} = \|_{i=1..4}(\mathit{i\_LiftCtrl} \parallel \mathit{i\_Lift})$ interacting with the $\mathit{DispatchCtrl} \parallel \mathit{Dispatch}$ component, and with all events apart from *req* and *bottom* internal:

$$(\|_{i=1..4}(\mathit{i\_LiftCtrl} \parallel \mathit{i\_Lift}) \parallel (\mathit{DispatchCtrl} \parallel \mathit{Dispatch})) \setminus \mathit{Int}$$

$$\mathit{Int} = \bigcup_i \{| \mathit{i\_inc}, \mathit{i\_dec}, \mathit{i\_isZero}, \mathit{i\_up}, \mathit{i\_down}, \mathit{i\_ground}\} \cup \{| \mathit{send}, \mathit{reset} |\}$$

We will see in Section 6 that this system is deadlock-free and divergence-free.

4 Deadlock-freedom

An essential requirement for controlled components is deadlock-freedom. This is easily checked in FDR, but only for processes that are expressed in CSP. Thus, we aim to establish a theorem that allows the deadlock-freedom of $P \parallel M$ to be deduced from deadlock-freedom of P (which can then be checked using FDR).

In general, parallel composition does not preserve deadlock-freedom. Fortunately, in the case of CSP controllers and B machines, we are able to identify conditions which ensure that the processes involved interact on their common channels in a particular way, ensuring that introducing a B machine cannot introduce any new deadlocks. In other words, any deadlocks possible for the controlled component $P \parallel M$ must already have been possible in P.

Open on possible inputs The required property of the B machine is that it should always be able to accept any input for any operation, and be able to provide some output. The need for this property is precisely why only machines with non-blocking operations are permitted. If a machine meets this property then we will say it is *open* on the particular operations and inputs.

In CSP terms, this is defined formally for CSP processes Q as follows:

Definition 2. *A process Q is* open *on a set of partial events PE if, given any $(tr, X) \in \mathcal{SF}[\![Q]\!]$ and $e \in PE$, there is some w such that $e.w \notin X$.*

This will apply to B machines as follows: given any machine operation $w \longleftarrow e(v)$, we would expect the machine to be open on any partial event of the form $e.v_0$, which corresponds to passing the input v_0 to operation e. In other words, there should be some output w_0 which is made available by the machine (and hence does not appear in the refusal set X).

The set of *possible inputs* for a machine will be all those partial events which correspond to operations being called with some input. The events are partial because they do not include the output values.

Definition 3. *Given a B machine M with operations $w_i \longleftarrow e_i(v_i)$, the set $pi(M)$ of possible inputs for M is defined by*

$$pi(M) = \bigcup_i \{e_i.v_i \mid v_i \in T_{in}(e_i)\}$$

Example 1. The set of partial inputs for the machine *i_Lift* is given in terms of the three operations as follows:

$$pi(i\_Lift) = \{|\ i\_inc.i \mid i \in \mathbb{Z}\ |\} \cup \{|\ i\_dec\ |\} \cup \{|\ i\_isZero\ |\}$$

Observe that in the cases of *i_inc* and *i_dec* there are no outputs, so the partial events are in fact complete events. Being open on these events means that they cannot be refused (since their output field is empty). There are two completions of the partial event *i_isZero*: *i_isZero.true* and *i_isZero.false*. *i_Lift* being open on this partial event means that at any stage at least one of these completions cannot be refused by *i_Lift*.

The key property of non-blocking machines is that they will always be open on their possible inputs:

Lemma 1. *Any (non-blocking) B machine M is open on $pi(M)$.*

This states in CSP semantics terms that any operation call with any input should always produce some result.

Non-discriminating controllers The condition on a controller P is that, whenever it calls an operation of the controlled B machine M, it should be able to accept any output provided by M. We call this property *non-discriminating*, and it can be expressed formally in CSP terms with the following definition:

Definition 4. *A CSP process P is* non-discriminating *on a set of partial events PE if, for any failure $(tr, X) \in \mathcal{SF}[\![P]\!]$ and subset $CV \subseteq PE$, we have that*

$$(\forall\, c.v \in CV \bullet \exists\, w \bullet c.v.w \in X) \Rightarrow (tr, X \cup \{|\ CV\ |\}) \in \mathcal{SF}[\![P]\!]$$

This definition states that if any event $c.v.w$ can be refused (i.e. appears in the refusal set X), then all the inputs on channel $c.v$ (i.e. outputs from the B machine) could be refused: thus the refusal X can be augmented with $\{|\ c.v\ |\}$.

Example 2. The control process *i_LiftCtrl* is non-discriminating on *i_isZero*: at any stage, *i_LiftCtrl* can either refuse all of $\{|\ i\_isZero\ |\}$, or else none of it. In terms of the definition, whenever some event from $\{i\_isZero.true, i\_isZero.false\}$ can be refused, then all can be refused.

Observe that *i_LiftCtrl* is also non-discriminating on $\{i\_inc.i \mid i \in \mathbb{Z}\}$ and on *i_dec*. In fact a process will trivially be non-discriminating on complete events.

The approach is restricted to *non-blocking* B machines. In other words, operations $w \longleftarrow e(v)$ must always be enabled (though they might be called outside their preconditions, which leads to divergence) and on any input they must provide some output.

Controllers which do not include blocking assertions on the control channels are able to accept any output from the associated B machine whenever they call an operation with any particular inputs. Thus, they will be non-discriminating on the possible inputs to the machine. This is expressed by the following lemma:

Lemma 2. *If P is a controller for machine M with no blocking assertions on any channels of M, then P is non-discriminating on the set $pi(M)$ of M's possible inputs.*

Observe that this lemma is illustrated by *i_LiftCtrl* in Example 2 above.

Establishing Deadlock-freedom We now have ingredients which are sufficient to deduce deadlock-freedom of $P \parallel Q$ from deadlock-freedom of P. The idea is that the interface between P and Q is defined by a set of partial events PE: P should be non-discriminating on these partial events, and Q should be open on them. We can show that if $P \parallel Q$ can deadlock, then so can P.

If $P \parallel Q$ does have a deadlock state, then all events can be simultaneously refused in that state. For any partial event e, Q is open on e so Q cannot refuse all of $\{| e |\}$. Hence P must be refusing some event in $\{| e |\}$, and so because P is non-discriminating, P can refuse all of $\{| e |\}$. Thus, we find that all events in the interface can be refused by P in this state, and P cannot perform any other events either. Hence P is in a deadlocked state.

Consider this reasoning in the context of a controlled component. Consider a state of $P \parallel M$. If P in this state is not deadlocked, then either

1. P is ready to perform an event outside $\alpha(M)$. In this case, M cannot prevent that event, and the combination $P \parallel M$ is ready to perform the event, and hence is not deadlocked; or
2. P is ready to perform an interaction with M. In this case, it is an operation call c with some input v. P is ready to accept any output from this operation call, since it is non-discriminating on $c.v$. M is ready to provide an output w in response to $c.v$, since it is open on $c.v$. Hence, the combination $P \parallel M$ is ready to perform $c.v.w$, and so is not in a deadlocked state.

The lemma that this reasoning establishes is the following:

Lemma 3. *If*

1. *P is non-discriminating on a set of partial events PE; and*
2. *Q is open on PE; and*
3. *$\alpha(Q) = \{| PE |\}$;*

then: if P is deadlock-free in the stable failures model, then so too is $P \parallel Q$

For a particular controlled component $P \parallel M$, we already have the conditions for Lemma 3: P is non-discriminating on $pi(M)$ (from Lemma 2); M is open on $pi(M)$ (from Lemma 1); and $\alpha(M) = \{| pi(M) |\}$.

Finally, we obtain the following theorem for controlled components:

Theorem 2. *If P is a CSP controller for M with no blocking assertions on any channels of M, and P is deadlock-free in the stable failures model, then $P \parallel M$ is deadlock-free in the stable failures model.*

This theorem is exactly what is required to establish deadlock-freedom of $P \parallel M$ from deadlock-freedom of P. In fact a direct proof of this theorem in terms of the CSP semantics has previously been presented, in [ST02b]. However, we find the identification of the properties non-discriminating and open yields more understanding as to why the theorem works and allows an easier proof of Theorem 2 and others.

Example 3. For example, consider the combination $i\_LiftCtrl \parallel i\_Lift$, in a state after some trace tr, in which $\{i\_isZero.true, i\_isZero.false\}$ is refused. We know that i_Lift is open on $\{| \ i\_isZero \ |\}$, so it cannot refuse the whole set $\{i\_isZero.true, i\_isZero.false\}$. Since the parallel combination does refuse that whole set, it must be that $i\_LiftCtrl$ is refusing at least one of $i\_isZero.true$, $i\_isZero.false$. But $i\_LiftCtrl$ is non-discriminating on i_isZero, so this means that it can itself refuse the whole set $\{| \ i\_isZero \ |\}$.

The same reasoning applies to all partial events in the interface between $i\_LiftCtrl$ and i_Lift. Thus, if $i\_LiftCtrl \parallel i\_Lift$ could reach a deadlock state, then all events in the interface would be refused by $i\_LiftCtrl \parallel i\_Lift$, and so they could also be refused purely by $i\_LiftCtrl$. Thus, $i\_LiftCtrl$ would also have a deadlock state.

As observed previously, $i\_LiftCtrl$ is deadlock-free. Hence Theorem 2 allows us to deduce that $i\_LiftCtrl \parallel i\_Lift$ is deadlock-free.

5 Restricting events to prevent divergence

The use of abstraction is essential in the compositional development of large systems. We will therefore generally need to hide control channels within controlled components.

Since hiding has the potential to introduce divergence, we need to be able to establish when this does not occur. In particular, it would be useful to be able to check divergence-freedom of a controller $P \setminus C$ using FDR, and to be able to deduce divergence-freedom of the controlled component $(P \parallel M) \setminus C$.

The following theorem on CSP processes P and Q gives such a condition:

Theorem 3. *If $P \parallel Q$ is divergence-free, and $C \subseteq \alpha(P)$, and $P \setminus C$ is divergence-free, then $(P \parallel Q) \setminus C$ is divergence-free.*

This is immediately applicable to controlled components, since $C \subseteq \alpha(P)$ as a consequence of our architecture. Thus, divergence-freedom of $(P \parallel M) \setminus C$ follows directly from divergence-freedom of $P \setminus C$.

However, in practice it will often be the case that $P \setminus C$ turns out not to be divergence-free, even if $(P \parallel M) \setminus C$ is. For instance, in the lift example we found that $i\_LiftCtrl \setminus \{| \ inc, dec, isZero \ |\}$ was not divergence-free, and instead we had to transform the controller description to $i\_LiftCtrl3(0)$ in order to obtain a controller such that $i\_LiftCtrl3(0) \setminus \{| \ inc, dec, isZero \ |\}$ is divergence-free. So it is necessary to identify theorems which justify such transformations.

Our approach is to identify behaviours of controller P which cannot occur in the context of the machine M under control. We then aim to find P' such that

1. P' is the same as P except (possibly) on the behaviours that have been identified, and
2. $P' \setminus C$ is divergence-free

Thus, $P' \parallel M$ will be the same as $P \parallel M$, which by assumption is divergence-free. Theorem 3 applied to P' yields that $(P' \parallel M) \setminus C$ is divergence-free, and hence $(P \parallel M) \setminus C$ is divergence-free.

This is the approach that was taken in the lift example. The relevant behaviour that cannot occur in the context of *i_Lift* is the output of *false* from *isZero* when the lift is at the ground floor. This behaviour is blocked in $i\_LiftCtrl3(0)$. However, $i\_LiftCtrl3(0)$ is the same as *i_LiftCtrl* for all behaviours that are possible in parallel with *i_Lift*.

The way we identify traces that cannot occur is to require divergence whenever they do occur, and then look for divergences. If we are concerned with a set of traces $T \subseteq A^*$, then we can express this by defining a new process $DIV_A(T)$ which behaves as RUN_A except that it diverges on any trace in T:

$$\mathcal{F}\,[\![DIV_A(T)]\!] = \{(tr, \{\}) \mid tr \in A^*\} \cup \{(tr \frown tr', X) \mid tr \in T \wedge tr' \in A^* \wedge X \subseteq A\}$$
$$\mathcal{D}\,[\![DIV_A(T)]\!] = \{tr \frown tr' \mid tr \in T \wedge tr' \in A^*\}$$

Observe that $DIV_A(\{\}) = RUN_A$ and $DIV_A(A^*) = DIV_A$.

The process $DIV_A(T)$ can then be used to mask behaviour in a process P. The process $P \parallel DIV_A(T)$ behaves exactly as P, except that whenever a trace in T is performed then it diverges. Thus, if $P \parallel DIV_A(T) = P' \parallel DIV_A(T)$, then P and P' have the same behaviour except possibly with regard to traces in T, which are masked by the introduction of divergence.

The following theorem allows a process P to be replaced by an alternative process P' in the context of another process Q. In particular, if P does not diverge in the context of Q (i.e. $P \parallel Q$ is divergence-free), and P' is the same as P except on divergent traces of P, then P and P' have the same executions when executed in parallel with Q (since none of P's divergent traces will be performed).

Theorem 4. *If P, P' and Q are such that*

1. *$P \parallel Q$ is divergence-free,*
2. *$P =_{FD} P' \parallel DIV_{\alpha(P)}(\mathcal{D}\,[\![P]\!])$*
3. *$\alpha(P) = \alpha(P')$*

then $P \parallel Q = P' \parallel Q$.

This states that if P' is different to P only with respect to where P diverges, and $P \parallel Q$ does not diverge, then P and P' behave the same in the context of Q. This follows because if $P \parallel Q$ does not diverge, then none of the traces of P which lead to divergence are possible when executing in parallel with Q. Since P' is exactly the same as P except for these traces, and Q prevents such traces from occurring, it follows that $P' \parallel Q$ is the same as $P \parallel Q$.

Example 4. As an example to illustrate Theorem 4, consider the following processes. P and P' have alphabet $A = \{a, b, c\}$, and Q has alphabet $\{a, b\}$.

$$P = (a \to (b \to DIV_A \,\Box\, a \to c \to P))$$
$$P' = (a \to (b \to c \to P' \,\Box\, a \to c \to P'))$$
$$Q = (a \to a \to Q) \,\Box\, (b \to STOP)$$

- Firstly, we see that $P \parallel Q$ can only ever perform a and c events, and is deadlock-free. In particular, the process Q prevents P from performing the b event, the only event that can lead to divergence, since there is no point at which P and Q can agree to perform b.
- The behaviour of P' after b occurs is different to that of P (which is divergent), but if b does not occur then P and P' behave the same. Thus, P and P' are the same except on the divergences of P.
- Finally, note that P and P' have the same alphabet.

Thus, we can conclude that $P \parallel Q = P' \parallel Q$.

The reason this result is useful is because it supports the introduction and manipulation of assertions on the control channels. If we introduce a divergent assertion on a control channel between P and M, and we then establish that $P \parallel M$ is divergence-free (using CLI techniques), then we can alter the behaviour of P when the assertion is false (in which case P diverges) and obtain a related controller P' which matches P outside P's divergences, and for which $P \parallel M = P' \parallel M$. The aim is to obtain a controller P' in this way for which $P' \setminus C$ is divergence-free.

The next lemma lists some ways in which diverging assertions within a controller can be transformed.

Lemma 4. *If a controller P' is obtained from controller P by replacing clauses of the form $e!v?x\{E(x)\} \to R(x)$ with one of:*

1. $e!v?x\{E'(x)\} \to R(x)$ *where* $\forall x. E(x) \Rightarrow E'(x)$
2. $e!v?x \to \mathsf{if}\ E(x)\ \mathsf{then}\ R(x)\ \mathsf{else}\ Q(x)$
3. $e!v?x \to R(x)$
4. $e!v?x\langle E(x)\rangle \to R(x)$

then $P =_{FD} P' \parallel DIV_{\alpha(P)}(\mathcal{D}\,[\![P]\!])$

Thus, we obtain the following corollary for controlled components:

Corollary 1. *If $P \parallel M$ is divergence-free, then behaviour in P following an input which fails a diverging assertion can be changed in accordance with Lemma 4 without affecting the behaviour of the parallel combination.*

This means that diverging assertions in P, once they have been discharged in a context M, can be replaced with blocking assertions, or else removed completely. This is precisely the justification for the transformation of $i\_LiftCtrl2(i)$ to $i\_LiftCtrl3(i)$: in the context of i_Lift, $i\_LiftCtrl2(0)$ does not diverge.

6 Parallel combinations of controlled components

All the results of the previous sections have been presented as applying to a single CSP controller process P in parallel with a single B machine M. However, systems we are generally concerned with (such as the combination of lifts) have the form $\big\|_i (P_i \parallel M_i)$, as illustrated in Figure 1. Many of the results we have obtained for a single controlled component can be lifted to combinations of components, and we will consider some of these in this section.

Divergence-freedom Firstly, we consider divergence-freedom. It is straightforward to establish divergence-freedom of a combined system, using the following theorem from [ST02b]:

Theorem 5. *If $P_i \parallel M_i$ are divergence-free for each i, then $\big\|_i (P_i \parallel M_i)$ is divergence-free.*

This follows immediately from the semantics for parallel composition, which preserves divergence-freedom. Thus, we need only establish divergence-freedom for the component pairs, and the result follows.

Example 5. In the parallel lift system, since each of the controlled lift components is divergence-free, and since we are given that the controlled dispatcher component is divergence-free, it follows that the overall parallel combination of all the components of the multiple lift system is divergence-free.

Establishing deadlock-freedom Associativity and commutativity of the parallel operator means that we can group the controller processes together and the machines together, rearranging the parallel composition as follows:

$$\big\|_i (P_i \parallel M_i) = (\big\|_i P_i) \parallel (\big\|_i M_i)$$

Now we can consider $(\big\|_i P_i)$ as a CSP process, and $(\big\|_i M_i)$ as another CSP process; and we are concerned with the parallel combination of these two processes.

The reason for grouping the components in this way is that the properties 'non-discriminating' and 'open' are preserved by parallel composition in CSP.

We can thus obtain the following two lemmas:

Lemma 5. *If P_i is a collection of controllers for machines M_i respectively, where each P_i has no blocking assertions on any channels of its associated M_i, then $\big\|_i P_i$ is non-discriminating on the set $\bigcup_i (pi(M_i))$.*

Lemma 6. *Any collection of (non-blocking) B machines M_i has that $\big\|_i M_i$ is open on $\bigcup_i (pi(M_i))$.*

Lemma 6 states that if each machine is able to engage in any of its operations, then the parallel combination of all the machines is able to engage in any of the operations of any of its machines.

These two lemmas mean that the conditions for Lemma 3 are met for controllers with no blocking assertions:

1. $\big\|_i P_i$ is non-discriminating on the set $\bigcup_i (pi(M_i))$.
2. $\big\|_i M_i$ is open on $\bigcup_i (pi(M_i))$.
3. $\alpha(\big\|_i M_i) = \{\mid \bigcup_i (pi(M_i)) \mid\}$.

This means that Lemma 3 is directly applicable to a collection of parallel controlled components, in which deadlock-freedom of the overall parallel combination follows from deadlock-freedom of the combination of controllers.

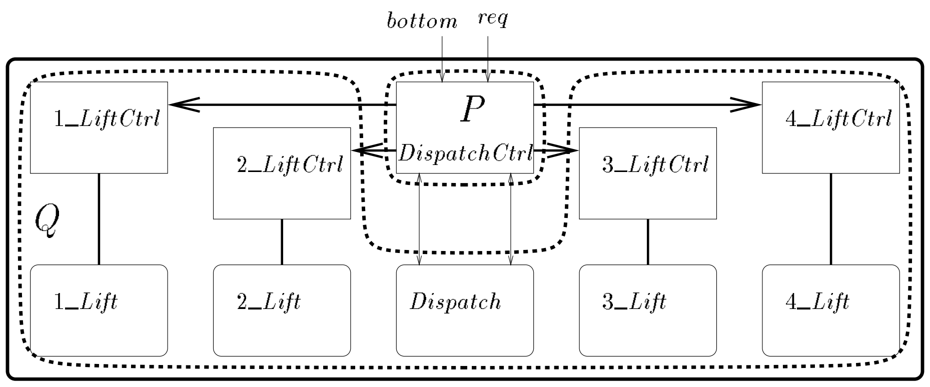

Fig. 7. Splitting the system into P and Q to verify divergence-freedom

Theorem 6. *Given a collection of CSP controllers P_i and corresponding controlled machines M_i, such that no controller has any blocking assertions on the control channels: then if $\big\|_i P_i$ is deadlock-free in the stable failures model, then so too is $\|_i (P_i \| M_i)$.*

In the example lift system, we have therefore only to check that

$(\big\|_{i=1..4} i\_LiftCtrl) \| DispatchCtrl$

is deadlock-free (which is easily shown) to deduce this for the complete system.

Divergence-freedom of Lift System We are really concerned with divergence-freedom of

$(\big\|_{i=1..4}(i\_LiftCtrl \| i\_Lift) \| (DispatchCtrl \| Dispatch)) \setminus Int$

Theorem 3 is the appropriate theorem to apply here. We need to split the system into P and Q such that $P \| Q$ is divergence-free, and $P \setminus C$ is divergence-free. The natural approach would take P as the combination of CSP controllers, and Q as the combination of B machines; verification could indeed be established by introducing assertions into the controllers along the lines of Section 3.

However, we have already established the individual lifts are divergence-free, so we can re-use this result by splitting the system differently, as pictured in Figure 7. P is $DispatchCtrl$, Q is the rest of the system, and C is the interface between P and Q:

$$\begin{aligned} P &= DispatchCtrl \\ Q &= \big\|_i i\_LiftSys \| Dispatch \\ C &= \bigcup_i \{| \, i\_up, i\_down, i\_ground \, |\} \cup \{| \, send, reset \, |\} \end{aligned}$$

We can check the conditions for Theorem 3:

1. Each *i_LiftSys* is divergence-free (as established earlier), and also *DispatchCtrl* $\|$ *Dispatch* is divergence-free, so the parallel combination $P \parallel Q = \big\|_i \, i\_LiftSys \parallel Dispatch \parallel DispatchCtrl$ is divergence-free (since divergence-freedom is preserved by parallel composition).
2. $C \subseteq \alpha(P)$
3. $P \setminus C$ is divergence-free. (This is easily checked with FDR.)

Thus $Lifts = (P \parallel Q) \setminus C$ is divergence-free.

7 Discussion

This paper has been concerned with providing the CSP underpinnings for developing controlled components consisting of B machines controlled by CSP controllers under a particular architecture. The work builds on the *control loop invariant* method for verifying individual controlled components in the context of the B Method, and develops results for combining such verified components.

All of the results presented in this paper have been developed using the CSP semantics of all the component processes. The emphasis has been on obtaining compositional results which enable existing CSP verification methods and tools to apply to our combined systems. These results enable a particular strategy for verification: transform system descriptions to equivalent forms which are amenable to CSP checking. In the simplest case, if the combination $P \parallel M$ is equivalent to $P' \parallel M$, and properties of $P' \parallel M$ can be established by analysing P' (with CSP tools), then those same properties can be deduced for $P \parallel M$. So our approach is to transform a controller P to a process P' which behaves the same way in the context of M.

Transforming system descriptions to enable pure CSP analysis may involve the introduction of state information within the CSP controller descriptions, so that the behaviour in the context of the underlying B machine is not affected. In this paper we have illustrated the use of this technique.

Ongoing work [ST02a] has obtained further results for this framework. Firstly, it is often the case that controlled components are only correct in the context of the rest of the system. In this situation we will need to introduce assertions on the channels between CSP controllers, in order to establish divergence-freedom of the individual controlled components. Treating assertions as blocking or diverging in particular cases is a delicate issue and depends on the particular verification under consideration. We have developed theorems [ST02a] which justify the use of particular kinds of assertions. Secondly, we have results (whose proofs use the notions of non-discriminating and open) concerning refinement in the stable failures model: if $SPEC \sqsubseteq P \setminus \alpha(M)$ then $SPEC \sqsubseteq (P \parallel M) \setminus \alpha(M)$ under the appropriate conditions. This enables specified properties to be verified of combined systems. These results have been applied to a Bounded Retransmission Protocol [EST03] for buffer-style properties, and in the Bank case study[TSB03].

There are several other approaches to combining a process-style controller with a state-based system description (e.g. [But00,FL03,WC01,SD01]). The approach closest to ours is Butler's csp2B tool [But00], which allows a CSP process to be conjoined to a B machine in a way which corresponds to a controller for an underlying machine. However, none of the other approaches exploit the semantic models for CSP in the way presented here. The ability to develop theory and tap into existing tool support on both the concurrency side and the state-based side is an important driver of the approach presented in this paper, and originally motivated the choices of CSP and B as the methods we chose to integrate.

Acknowledgements Thanks are due to Neil Evans, Susan Stepney, Fiona Polack and Régine Laleau for discussions on this work, and also to Neil for comments on drafts of this paper.

References

[But00] M. Butler. csp2B: A practical approach to combining CSP and B. *Formal Aspects of Computing*, 12, 2000.

[EST03] N. Evans, S. A. Schneider, and H. E. Treharne. Investigating a file transmission protocol using CSP and B. In *proceedings of ST.EVE workshop*, 2003.

[FL03] M. Frappier and R. Laleau. Proving event ordering properties for information systems. In *ZB2003*, 2003.

[For97] Formal Systems (Europe) Ltd. *Failures-Divergences Refinement: FDR2 Manual*, 1997.

[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 J. Dijkstra*. Springer-Verlag, 1990.

[Ros97] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.

[Sca98] B. Scattergood. *The Semantics and Implementation of Machine-Readable CSP*. D. Phil thesis, Oxford University, 1998.

[Sch99] S.A. Schneider. *Concurrent and Real-time Systems: The CSP approach*. Wiley, 1999.

[SD01] G. Smith and J. Derrick. Specification, refinement and verification of concurrent systems - an integration of Object-Z and CSP. *Formal Methods in System Design*, 18(3), 2001.

[ST02a] S. Schneider and H. Treharne. CSP theorems for communicating B machines. Technical Report CSD-TR-02-05, Royal Holloway, University of London, 2002.

[ST02b] S.A. Schneider and H.E. Treharne. Communicating B machines. In *ZB2002*, volume LNCS 2272, 2002.

[Tre00] H. E. Treharne. *Combining control executives and software specifications*. PhD thesis, Royal Holloway, University of London, 2000.

[TSB03] H.E. Treharne, S.A. Schneider, and M. Bramble. Combining specifications using communication. In *ZB2003*, 2003.

[WC01] J. C. P. Woodcock and A. L. C. Cavalcanti. A concurrent language for refinement. In *5th Irish Workshop on Formal Methods*, 2001.