CSP Theorems for Communicating B Machines

Steve Schneider and Helen Treharne

Technical Report

CSD-TR-02-12

August 26, 2003

Department of Computer Science
Egham, Surrey TW20 0EX, England

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 requirement 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 [ST02] to restrict the interaction between components, by ensuring that each B machine interacts only with its own controller. A system will be structured as 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

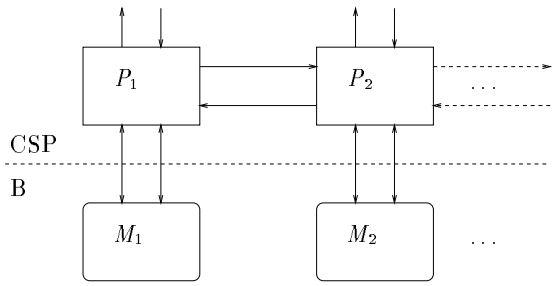

Figure 1 A CSP and B combined system architecture

particular, both CSP and B already have mature tool support which can be used to verify the components.

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.

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.

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.

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

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 [ST02].

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

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 is a data variable, v is a data value, $E(x)$ 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. The process expression $S(p)$ expresses a recursive call. Finally, processes can be defined using (recursive) definitions of the form $S(p) \mathrel{\widehat{=}} P$.

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 $\mathcal{SF}[\![P]\!]$ 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 $\mathcal{SF}[\![P]\!] = \mathcal{SF}[\![Q]\!]$ 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 $\mathcal{D}[\![P]\!]$.

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, which 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 generally easier and more efficient, and to establish results which remain valid in the failures-divergences model. For example, once it has been established that 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. There are also CSP theorems (for example, Theorem 3 in this paper) for establishing that a process P is divergence-free.

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 $\neg wp(e_1;\ e_2;\ \ldots;\ e_n, false)$, or in Abstract Machine Notation $\neg([e_1;\ e2;\ \ldots;\ e_n]false)$. (The empty trace is treated as *skip*).

```
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
```

$i\_LiftCtrl \mathrel{\widehat=}$
$\quad i\_up?y \rightarrow i\_inc!y \rightarrow i\_LiftCtrl$
$\quad \Box\ i\_down?y \rightarrow i\_DOWN(y)$
$\quad \Box\ i\_ground \rightarrow i\_LOWER$

$i\_DOWN(n) \mathrel{\widehat=}$
$\quad \mathsf{if}\ n = 0$
$\quad \mathsf{then}\ i\_LiftCtrl$
$\quad \mathsf{else}\ i\_isZero?bb \rightarrow$
$\qquad \mathsf{if}\ (bb = TRUE)$
$\qquad \mathsf{then}\ i\_LiftCtrl$
$\qquad \mathsf{else}\ i\_dec \rightarrow i\_DOWN(n-1)$

$i\_LOWER \mathrel{\widehat=}$
$\quad i\_isZero?bb \rightarrow$
$\qquad \mathsf{if}\ (bb = TRUE)$
$\qquad \mathsf{then}\ i\_LiftCtrl$
$\qquad \mathsf{else}\ i\_dec \rightarrow i\_LOWER$

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

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. $\neg([e_1;\ e2;\ \ldots;\ e_n]true)$. 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:*

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

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

We have a technique [Tre00,ST02], 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.

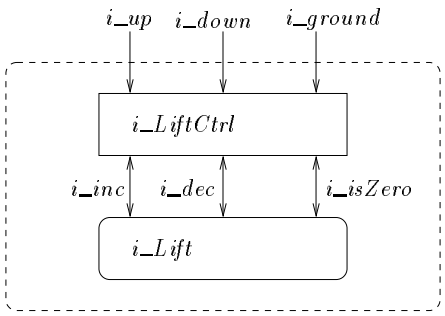

Figure 3 The controlled lift system

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. We will indicate the use of the theorems presented later in the paper. 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.

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

$$i\_LiftSys \mathrel{\widehat=} (i\_Lift \parallel i\_LiftCtrl) \setminus \{\!|\, i\_inc, i\_dec, i\_isZero \,|\!\}$$

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, so instead we 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 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 [ST02] 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 in Section 4.

We also establish in this paper (Theorem 3 in Section 5) 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 $i\_LiftCtrl \setminus \{|\ i\_inc, i\_dec, i\_isZero\ |\}$ 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 $i\_LiftCtrl \setminus \{|\ i\_inc, i\_dec, i\_isZero\ |\}$ 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. we do this by introducing a new variable f, and also introducing 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

$i\_LiftCtrl2(f) \mathrel{\widehat{=}}$
$\quad i\_up?y \rightarrow i\_inc!y \rightarrow i\_LiftCtrl2(f+y)$
$\quad \Box\ i\_down?y \rightarrow i\_DOWN2(f,y)$
$\quad \Box\ i\_ground \rightarrow i\_LOWER2(f)$

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

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

Figure 4 The controller with diverging assertions

$i\_LiftCtrl3(f) \mathrel{\widehat{=}}$
$\quad i\_up?y \rightarrow i\_inc!y \rightarrow i\_LiftCtrl3(f+y)$
$\quad \Box\ i\_down?y \rightarrow i\_DOWN3(f,y)$
$\quad \Box\ i\_ground \rightarrow i\_LOWER3(f)$

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

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

Figure 5 The controller with blocking assertions

in Figure 4. It is straightforward to show that $i\_LiftCtrl2(0)$ is an appropriate driver for i_Lift (using control loop invariant $f = i\_floor$ which relates the CSP state to the state of the B machine). 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 $i\_LiftCtrl2(0) \setminus \{|\ i\_inc, i\_dec, i\_isZero\ |\}$ for divergence-freedom. However, in the context of i_Lift we know the assertion will always be 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 (given at the end of Section 5). Thus, we obtain a variant $i\_LiftCtrl3(0)$ of the controller, given in Figure 5, such that $i\_LiftCtrl3(0) \parallel i\_Lift =_{FD} i\_LiftCtrl2(0) \parallel i\_Lift$.

$i\_LiftCtrl4(f) \mathrel{\widehat{=}}$
$\quad i\_up?y \rightarrow i\_inc!y \rightarrow i\_LiftCtrl4(f+y)$
$\quad \Box\ i\_down?y \rightarrow i\_DOWN4(f, y)$
$\quad \Box\ i\_ground \rightarrow i\_LOWER4(f)$

$i\_LOWER4(f) \mathrel{\widehat{=}}$
$\quad i\_isZero?bb \rightarrow$
$\qquad$ if $(bb = TRUE)$
$\qquad$ then $i\_LiftCtrl4(f)$
$\qquad$ else $i\_dec \rightarrow i\_LOWER4(f-1)$

$i\_DOWN4(f, n) \mathrel{\widehat{=}}$
$\quad$ if $n = 0$
$\quad$ then $i\_LiftCtrl4(f)$
$\quad$ else $i\_isZero?bb \rightarrow$
$\qquad$ if $(bb = TRUE)$
$\qquad$ then $i\_LiftCtrl4(f)$
$\qquad$ else $i\_dec \rightarrow$
$\qquad\quad i\_DOWN4(f-1, n-1)$

Figure 6 The controller with all assertions dropped

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 $i\_LiftCtrl4(0)$, whose behaviour does not depend on the value of the parameter f at all. This controller is given in Figure 6. Theorem 11 in Section 8 yields that $i\_LiftCtrl(0)$ is equivalent to $i\_LiftCtrl$. We have therefore finally 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.

$$i\_LiftCtrl2(0) \parallel i\_Lift =_{FD} i\_LiftCtrl3(0) \parallel i\_Lift =_{FD} i\_LiftCtrl4(0) \parallel i\_Lift$$

- ◇ The combination $i\_LiftCtrl2(0) \parallel i\_Lift$ can be shown to be divergence-free using techniques from [ST02].
- ◇ $i\_LiftCtrl3(0) \setminus \{|\ i\_inc, i\_dec, i\_isZero\ |\}$ is divergence-free, and so $(i\_LiftCtrl3(0) \parallel i\_Lift) \setminus \{|\ i\_inc, i\_dec, i\_isZero\ |\}$ is divergence-free.
- ◇ And $i\_LiftCtrl4(0) \parallel i\_Lift$ 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 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 $i\_LiftCtrl \parallel i\_Lift$.

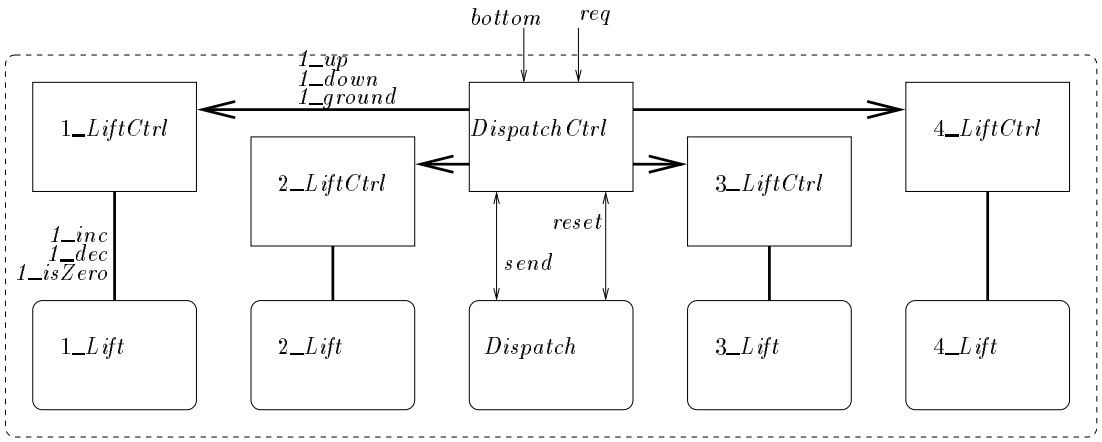

Figure 7 The complete system *Lifts*

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 7.

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:

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

Our overall system is then composed of the controlled lift components $Lifts \mathrel{\widehat=} \|_{i=1..4}(i\_LiftCtrl \parallel i\_Lift)$ interacting with the $DispatchCtrl \parallel Dispatch$ compo-

nent, and with all events apart from *req* and *bottom* internal:

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

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

We will see in Section 7 that this system is deadlock-free and divergence-free.

4 Deadlock-freedom

This section introduces two new properties concerning process behaviour on channels: *open on possible inputs*, and *non-discriminating*. These are the key properties exhibited by B machines and CSP controllers respectively. As we shall see, considering components in terms of these properties enables many of the results from Sections 4 and 5 concerning individual controlled components to be lifted to interacting collections of controlled components in Section 7. They also enable easier proofs of previously established results such as Theorem 2 in this section.

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 possible 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.

Our 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.

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.

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.*

Proof. By structural induction on P.

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. *$\alpha(Q) = \{\mid PE \mid\}$;*

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

Proof. We prove this result by contradiction.

Assume that there is some deadlock $(tr, \Sigma) \in \mathcal{SF}[\![P \parallel Q]\!]$, where $\Sigma = \alpha(P) \cup \alpha(Q)$. Then there must be refusal sets X_P and X_Q such that $X_P \cup X_Q = \Sigma$, with $(tr \upharpoonright \alpha(P), X_P) \in \mathcal{SF}[\![P]\!]$ and $(tr \upharpoonright \alpha(Q), X_Q) \in \mathcal{SF}[\![Q]\!]$. Then $\Sigma \setminus \alpha(Q) \subseteq \Sigma \setminus X_Q \subseteq X_P$, and so $\Sigma = X_P \cup \alpha(Q)$.

Now Q is open on PE so for each $e \in PE$ there is some w such that $e.w \notin X_Q$. Since $e.w \in \Sigma = X_Q \cup X_P$, it follows that $e.w \in X_P$. Since P is non-discriminating on PE it follows that $(tr \upharpoonright \alpha(P), X_P \cup \{\mid PE \mid\}) \in \mathcal{SF}[\![P]\!]$. But $\{\mid PE \mid\} = \alpha(Q)$, and so $(tr \upharpoonright \alpha(P), \Sigma) \in \mathcal{SF}[\![P]\!]$, since $\Sigma = X_P \cup \alpha(Q)$. Thus P has a deadlocking trace, contradicting the assumption that P is deadlock-free.

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 $\alpha(M) = \{\mid pi(M) \mid\}$.

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.*

Proof. This follows from Lemma 1, Lemma 2, and Lemma 3, earlier in this section, by observing that P is non-discriminating on $pi(M)$ and M is open on $pi(M)$.

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 [ST02]. 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. In the lift component example in Section 3, the channels i_inc, i_dec, and i_isZero are hidden, leaving i_up, i_down, and i_ground as the only external channels.

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.*

Proof. Assume for a contradiction that $tr_0 \in \mathcal{D}[\![(P \parallel Q) \setminus C]\!]$.

Then either $tr_0 = tr_1 \setminus C$ where $tr_1 \in \mathcal{D}[\![P \parallel Q]\!]$, contradicting the fact that $P \parallel Q$ is divergence-free; or $\exists\, tr_1, tr_2 \bullet tr_0 = tr_1 \setminus C \frown tr_2 \wedge \forall\, n \bullet \exists\, tr' \in C^* \bullet \#tr' > n \wedge tr_1 \frown tr' \in \mathcal{T}[\![(P \parallel Q)]\!]\}$. In this case we have that $\forall\, n \bullet \exists\, tr' \in C^* \bullet \#tr' > n \wedge tr_1 \upharpoonright \alpha(P) \frown tr' \in \mathcal{T}[\![P]\!]$. But this means that $tr_1 \upharpoonright \alpha(P) \in \mathcal{D}[\![P \setminus C]\!]$, contradicting the fact that $P \setminus C$ is divergence-free.

This is immediately applicable to controlled components (where the machine M is considered as the process Q) 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 $P \parallel M$. We are assuming that $P \parallel M$ has previously been shown to be divergence-free: that P is an appropriate controller for M. 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 $DIV_A(\{\}) =_{FD} RUN_A$ and $DIV_A(A^*) =_{FD} DIV_A$.

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 $P \parallel DIV_A(T) =_{FD} P' \parallel DIV_A(T)$, then P and P' have the same behaviour except possibly with regard to traces in T, which are masked by the introduction of divergence.

5.1 Equivalence on non-divergent behaviour

Given an upwards-closed set $T \subseteq A^*$ of traces (i.e. $tr \in T \Rightarrow tr \frown tr' \in T$) we can define a process $DIV_A(T)$ which behaves as RUN_A except that it diverges on any trace in T:

$$\begin{aligned}\mathcal{F}[\![DIV_A(T)]\!] &= \{(tr, \{\}) \mid tr \in A^*\} \cup T \times \mathbb{P}(\Sigma)\\ \mathcal{D}[\![DIV_A(T)]\!] &= T\end{aligned}$$

Observe that $DIV_A(\{\}) = RUN_A$ and $DIV_\Sigma(\Sigma^*) = DIV$.

The process $DIV_A(T)$ can 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.

Lemma 4. *For any process P:*

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

Proof. Let $R = DIV_{\alpha(P)}(\mathcal{D}[\![P]\!])$. We will prove that $P \parallel R$ has the same divergences and failures as P.

$$\begin{aligned}\mathcal{D}[\![P \parallel R]\!] &= \{tr \frown tr' \mid tr \in \mathcal{T}[\![P]\!] \wedge tr \in \mathcal{D}[\![R]\!]\}\\ &\quad \cup \{tr \frown tr' \mid tr \in \mathcal{D}[\![P]\!] \wedge tr \in \mathcal{T}[\![R]\!]\}\\ &= \{tr \frown tr' \mid tr \in \mathcal{D}[\![R]\!]\}\\ &\quad \cup \{tr \frown tr' \mid tr \in \mathcal{D}[\![P]\!]\}\\ &= \mathcal{D}[\![P]\!]\end{aligned}$$

$$\begin{aligned}\mathcal{F}[\![P \parallel R]\!] &= \{(tr, X_1 \cup X_2) \mid (tr, X_1) \in \mathcal{F}[\![P]\!] \wedge (tr, X_2) \in \mathcal{F}[\![R]\!]\}\\ &\quad \cup \{(tr, X) \mid tr \in \mathcal{D}[\![P \parallel R]\!]\}\\ &= \{(tr, X) \mid (tr, X) \in \mathcal{F}[\![P]\!]\}\\ &\quad \cup \{(tr, X) \mid tr \in \mathcal{D}[\![P]\!]\}\\ &= \mathcal{F}[\![P]\!]\end{aligned}$$

Lemma 5. *If $A \subseteq \alpha(P)$ and $P \parallel DIV_A(T)$ is divergence-free for some arbitrary upward-closed set of traces T, then $P = P \parallel DIV_A(T)$.*

Proof. We are given that $\mathcal{D}[\![P \parallel DIV_A(T)]\!] = \{\}$

$$\begin{aligned}\mathcal{D}[\![P \parallel DIV_A(T)]\!] &= \{tr \frown tr' \mid tr \in \mathcal{T}[\![P]\!] \wedge tr \upharpoonright A \in \mathcal{D}[\![DIV_A(T)]\!]\} \\ &\quad \cup \{tr \frown tr' \mid tr \in \mathcal{D}[\![P]\!] \wedge tr \upharpoonright A \in \mathcal{T}[\![DIV_A(T)]\!]\} \\ &= (\{tr \mid tr \in \mathcal{T}[\![P]\!] \wedge tr \upharpoonright A \in T\}) \cup \mathcal{D}[\![P]\!] \\ &= \{\}\end{aligned}$$

Hence $\mathcal{D}[\![P]\!] = \{\}$. Observe also that $\{tr \mid tr \in \mathcal{T}[\![P]\!] \wedge tr \upharpoonright A \in T\} = \{\}$.

$$\begin{aligned}\mathcal{F}[\![P \parallel DIV_A(T)]\!] &= \{(tr, X_1 \cup X_2) \mid (tr, X_1) \in \mathcal{F}[\![P]\!] \wedge (tr \upharpoonright A, X_2) \in \mathcal{F}[\![DIV_A(T)]\!]\} \\ &\quad \cup \{(tr, X) \mid tr \in \mathcal{D}[\![P \parallel DIV_A(T)]\!]\} \\ &= (\{(tr, X) \mid (tr, X) \in \mathcal{F}[\![P]\!] \wedge tr \upharpoonright A \notin T\} \\ &\quad \cup \{(tr, X) \mid tr \in \mathcal{T}[\![P]\!] \wedge tr \upharpoonright A \in T\}) \\ &\quad \cup \{\} \\ &= \mathcal{F}[\![P]\!]\end{aligned}$$

Hence $P \parallel DIV_A(T)$ and P agree on their failures and divergences, establishing the result.

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 $P \parallel Q =_{FD} P' \parallel Q$.

Proof. Let $D = DIV_{\alpha(P)}(\mathcal{D}[\![P]\!])$.

$$\begin{aligned}P \parallel Q &=_{FD} P' \parallel DIV_{\alpha(P)}(\mathcal{D}[\![P]\!]) \parallel Q \\ &=_{FD} P' \parallel Q\end{aligned}$$

The last step follows from Lemma 5 because $P' \parallel DIV_{\alpha(P)}(\mathcal{D}[\![P]\!]) \parallel Q$ is divergence-free.

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 $\{a, b\}$.

$$P \mathrel{\widehat{=}} (a \rightarrow (b \rightarrow DIV_A \,\Box\, a \rightarrow c \rightarrow P))$$
$$P' \mathrel{\widehat{=}} (a \rightarrow (b \rightarrow c \rightarrow P' \,\Box\, a \rightarrow c \rightarrow P'))$$
$$Q \mathrel{\widehat{=}} (a \rightarrow a \rightarrow Q) \,\Box\, (b \rightarrow STOP)$$

- ◇ 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 $P \parallel Q =_{FD} P' \parallel Q$.

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 $P \parallel M =_{FD} P' \parallel M$. 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 6. *If a controller P' is obtained from controller P by replacing clauses of the form $e!v?x\{E(x)\} \rightarrow R(x)$ with one of:*

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

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

Proof. By structural induction on the form of 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 6 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_LiftCtrl*2(i) to *i_LiftCtrl*3(i): in the context of *i_Lift*, *i_LiftCtrl*2(0) does not diverge.

6 Abstraction and refinement

In this section we consider the verification of controlled components with respect to refinement specifications. We will begin by considering traces refinement, where the results are straightforward. We will then consider stable failures refinement.

In the case of traces refinement, we immediately have the following result:

Lemma 7. *For any controller P and any B machine M we have that:*

$P \sqsubseteq_T (P \parallel M)$

Proof. This follows immediately from the trace semantics of parallel composition:

$$\begin{aligned}\mathcal{T}[\![P \parallel M]\!] &= (\mathcal{T}[\![P]\!] \cap \mathcal{T}[\![M]\!]) \\ &\subseteq \mathcal{T}[\![P]\!]\end{aligned}$$

This yields the following corollary.

Corollary 2.

1. *If $S \sqsubseteq_T P$ then $S \sqsubseteq_T (P \parallel M)$.*
2. *If $S \sqsubseteq_T P \setminus A$ then $S \sqsubseteq_T (P \parallel M) \setminus A$*

These follow from transitivity of refinement, and the second also uses monotonicity of the CSP operators (in this case hiding) with respect to refinement.

These results mean that it is sufficient to demonstrate a trace refinement $S \sqsubseteq_T P$ or $S \sqsubseteq_T P \setminus A$ purely on the CSP part of a controlled component, in order to deduce that it holds for the overall controlled component: $S \sqsubseteq_T P \parallel M$ or $S \sqsubseteq_T (P \parallel M) \setminus A$ respectively. In this way we can establish trace properties of controlled components.

When we consider stable failures, the situation is not so straightforward. In particular, a stable failures refinement of the form $S \sqsubseteq_{SF} P$ on a controller P can place liveness requirements on the interactions between P and its controlled machine. However, the introduction of the machine might violate the requirement even if P meets it. For example, if $S \mathrel{\widehat{=}} c?x \rightarrow S$ and $P \mathrel{\widehat{=}} c?x \rightarrow P$, then $S \sqsubseteq_{SF} P$. Yet if M is only prepared to perform $c.1$, and will block on $c.0$, then we find that $S \not\sqsubseteq_{SF} (P \parallel M)$.

Fortunately, we are able to obtain results in the case where the specification S is only concerned with the external events of P, and not the internal channels that P uses to interact with M. In this case we obtain the following theorem:

Theorem 5. *If*

- P *is non-discriminating on a set of partial events* PE*; and*
- Q *is open on* PE*; and*
- $\{\mid PE \mid\} \subseteq \alpha(P)$*; and*
- $\alpha(Q) = \{\mid PE \mid\}$*;*

then:

$$P \setminus \{\mid PE \mid\} \sqsubseteq_{SF} (P \parallel Q) \setminus \{\mid PE \mid\}$$

Proof. We aim to prove that

1. $\mathcal{T}[\![(P \parallel Q) \setminus \{\mid PE \mid\}]\!] \subseteq \mathcal{T}[\![P \setminus \{\mid PE \mid\}]\!]$; and
2. $\mathcal{SF}[\![(P \parallel Q) \setminus \{\mid PE \mid\}]\!] \subseteq \mathcal{SF}[\![P \setminus \{\mid PE \mid\}]\!]$

1. This is a case of Corollary 2 above.

2. Consider $(tr, X) \in \mathcal{SF}[\![(P \parallel Q) \setminus \{\mid PE \mid\}]\!]$. We aim to prove that $(tr, X) \in \mathcal{SF}[\![P \setminus \{\mid PE \mid\}]\!]$.

From the semantics of hiding there is some tr' such that $tr' \setminus \{\mid PE \mid\} = tr$ and $(tr', X \cup \{\mid PE \mid\}) \in \mathcal{SF}[\![P \parallel Q]\!]$. So there are X_P and X_Q such that:

- $(tr' \upharpoonright \alpha(P), X_P) \in \mathcal{SF}[\![P]\!]$;
- $(tr' \upharpoonright \alpha(Q), X_Q) \in \mathcal{SF}[\![Q]\!]$; and

$\diamond$ $X_P \cup X_Q = X \cup \{|\ PE\ |\}$.

Now $X_Q \subseteq \{|\ PE\ |\}$, and $X \cap \{|\ PE\ |\} = \{\}$, so $X \cap X_Q = \{\}$, and so $X \subseteq X_P$.

Now consider some $e \in PE$. There is some w such that $e.w \notin X_Q$, because Q is open on PE. However, $e.w \in \{|\ PE\ |\}$, and so $e.w \in X_P$.

Since this is true for each $e \in PE$, we obtain that $(tr', X_P \cup \{|\ PE\ |\}) \in \mathcal{SF}[\![P]\!]$ since P is non-discriminating on PE. It follows that

$(tr' \setminus \{|\ PE\ |\}, X_P \setminus \{|\ PE\ |\}) \in \mathcal{SF}[\![P \setminus \{|\ PE\ |\}]\!]$

Finally observe that $tr' \setminus \{|\ PE\ |\} = tr$ and $X_P \setminus \{|\ PE\ |\} = X$, since

$$\begin{aligned} X_P \setminus \{|\ PE\ |\} &= (X_P \cup X_Q) \setminus \{|\ PE\ |\} \\ &= (X \cup \{|\ PE\ |\}) \setminus \{|\ PE\ |\} \\ &= X \setminus \{|\ PE\ |\} \\ &= X \end{aligned}$$

establishing the result that $(tr, X) \in \mathcal{SF}[\![P \setminus \{|\ PE\ |\}]\!]$ as required.

Corollary 3. *If P is a CSP controller for M, and P has no guards on any channels of M, then $P \setminus \alpha(M) \sqsubseteq_{SF} (P \parallel M) \setminus \alpha(M)$.*

The following corollary of Theorem 5 means that it is sufficient to establish a stable failures refinement on $P \setminus \alpha(M)$ in order to deduce it for the controlled component $(P \parallel M) \setminus \alpha(M)$:

Corollary 4. *If $P \parallel M$ is a controlled component, then $S \sqsubseteq_{SF} P \setminus \alpha(M)$ and P has no guards on any channels of M then $S \sqsubseteq_{SF} (P \parallel M) \setminus \alpha(M)$*

Observe that all the above results require that the CSP controllers are non-blocking on the channels they use to communicate with their controlled components. Without this property, the result fails to hold. For example, if M is a machine that is always prepared to output the value 0 on channel *com*, expressed in CSP as:

$$M = com!0 \rightarrow M$$

and P is a controller that requires the value 1 on *com* (to pass on to external channel *out*) and blocks other values:

$$P = com?x\langle x = 1\rangle \rightarrow out!x \rightarrow P$$

Then $SPEC = \sqcap_x\ out!x \rightarrow SPEC$ has $SPEC \sqsubseteq_{SF} P \setminus \{|\ com\ |\}$, but $SPEC \not\sqsubseteq_{SF} (P \parallel M) \setminus \{|\ com\ |\}$, because it can deadlock.

Also observe that P is deadlock-free, but $P \parallel M$ can deadlock.

7 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 $\|_i (P_i \parallel M_i)$, 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 [ST02]:

Theorem 6. *If $P_i \parallel M_i$ are divergence-free for each i, then $\|_i (P_i \parallel M_i)$ 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:

$$\|_i (P_i \parallel M_i) =_{FD} (\|_i P_i) \parallel (\|_i M_i)$$

Now we can consider $(\|_i P_i)$ as a CSP process, and $(\|_i M_i)$ 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 obtain results concerning the non-discriminating nature of a parallel combination of CSP processes:

Theorem 7. *If PE is a set of partial events such that each P_i is non-discriminating on $PE \cap \alpha(P_i)$ then $\|_i P_i$ is non-discriminating on $PE \cap \bigcup_i \alpha(P_i)$.*

Proof. Consider $(tr, X) \in \mathcal{SF}[\![\|_i P_i]\!]$. Then there are refusal sets X_i such that $X = \bigcup_i X_i$, and $(tr \upharpoonright \alpha(P_i), X_i) \in \mathcal{SF}[\![P_i]\!]$ for each i.

Now consider $CV \subseteq PE$ such that $\forall\, c.v \in CV \bullet \exists\, w \bullet c.v.w \in X$. For each i, let $CV_i = \{c.v \mid \exists\, c.v.w \in X_i\}$. Then $CV = \bigcup_i CV_i$.

Now since $CV_i \subseteq PE \cap \alpha(P_i)$, and P_i is non-discriminating on $PE \cap \alpha(P_i)$, we have that $(tr \upharpoonright \alpha(P_i), X_i \cup \{|\ CV_i\ |\}) \in \mathcal{SF}[\![P_i]\!]$ for each i, and hence that $(tr, \bigcup_i (X_i \cup \{|\ CV_i\ |\})) \in \mathcal{SF}[\![\|_i P_i]\!]$, i.e. that $(tr, X \cup \{|\ CV\ |\}) \in \mathcal{SF}[\![\|_i P_i]\!]$, which completes the proof.

We obtain the following corollary:

Corollary 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 $\|_i P_i$ is non-discriminating on the set $\bigcup_i (pi(M_i))$.*

Proof. This follows from Lemma 2 and Theorem 7.

Lemma 8. *Any collection of (non-blocking) B machines M_i has that $\|_i M_i$ is open on $\bigcup_i (pi(M_i))$.*

Lemma 8 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. $\|_i P_i$ is non-discriminating on the set $\bigcup_i (pi(M_i))$.
2. $\|_i M_i$ is open on $\bigcup_i (pi(M_i))$.
3. $\alpha(\|_i M_i) = \{|\ \bigcup_i (pi(M_i))\ |\}$.

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.

Theorem 8. *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 $\|_i P_i$ is deadlock-free in the stable failures model, then so too is $\|_i (P_i \parallel M_i)$.*

Proof. This follows from Corollary 5, Lemma 8, and Lemma 3, by observing that $\|_i P_i$ is non-discriminating on $pi(\|_i M_i)$, and $\|_i M_i$ is open on $pi(\|_i M_i)$.

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

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

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

Observe that Theorem 8 applies to architectures in which machine operations can synchronise with a number of controllers. In other words, controllers can overlap on operations that they call. The theorem still requires that each machine M_i has its own controller P_i which is required to ensure consistency, but it allows other controllers P_j also to synchronise on such operation calls.

We are also able to lift the results from Section 6 to parallel combinations.

Corollary 6. *Given a collection of CSP controllers P_i and corresponding controlled machines M_i, such that no controller has any guards on the control channels: then*

$\|_i P_i \setminus (\bigcup_i \alpha(M_i)) \sqsubseteq_{SF} \|_i (P_i \parallel M_i) \setminus (\bigcup_i \alpha(M_i))$

This is a corollary of Theorem 5 together with Lemma 8 and Corollary 5.

Divergence-freedom of Lift System

We are really concerned with divergence-freedom of

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

Theorem 3 is the appropriate theorem to apply here. We need to split the system into P and Q such that $P \parallel Q$ 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 8. P is *DispatchCtrl*, Q is the rest of the system, and C is the interface between P and Q:

$$P \mathrel{\widehat{=}} DispatchCtrl$$
$$Q \mathrel{\widehat{=}} \|_i i\_LiftSys \parallel Dispatch$$
$$C = \bigcup_i \{| \ i\_up, i\_down, i\_ground \ |\} \cup \{| \ send, reset \ |\}$$

We can check the conditions for Theorem 3:

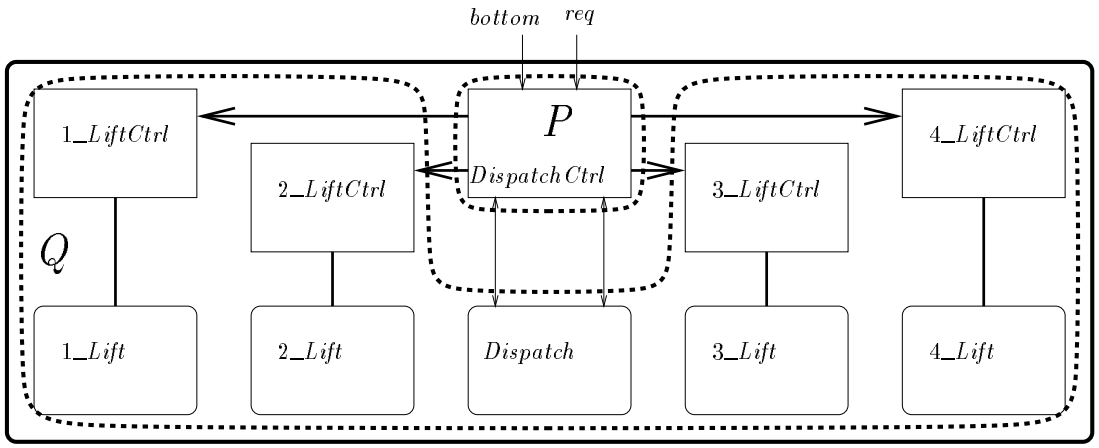

Figure 8 Splitting the system into P and Q to verify divergence-freedom

1. Each $i\_LiftSys$ is divergence-free (as established earlier), and also $DispatchCtrl \parallel Dispatch$ is divergence-free, so the parallel combination $P \parallel Q \mathrel{\widehat{=}} \|_i\, i\_LiftSys \parallel Dispatch \parallel DispatchCtrl$ 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 $Lifts \mathrel{\widehat{=}} (P \parallel Q) \setminus C$ is divergence-free.

7.1 Guards and assumptions: a toy example

Our approach is dependent on the ability to verify that individual controlled components $P_i \parallel M_i$ are divergence-free, and as mentioned previously this is done using the control loop invariant technique [Tre00,ST02]. This approach requires analysis of the controlled component in isolation.

However, when we consider the case of multiple concurrent controlled components, then correctness of any particular component might depend on the behaviour of the rest of the system. It is important to be able to incorporate relevant information about interactions between controllers into the analysis of individual controlled components. This is the reason for allowing guards and assumptions on the channels between controllers.

We will consider a toy example to illustrate the issues. Consider the machines Odd and $Even$ of Figure 9. Observe that the possible inputs of these machines are given by:

$$pi(Odd) = \{oddget\} \cup \{oddput.n \mid n \in \mathbb{N}\}$$
$$pi(Even) = \{evenget\} \cup \{evenput.n \mid n \in \mathbb{N}\}$$

```
MACHINE     Odd
VARIABLES   odd
INVARIANT   odd : NAT & odd mod 2 = 1
INITIALISATION  odd := 1
OPERATIONS
oddput(nn) = PRE nn : NAT & nn mod 2 = 1
             THEN odd := nn END;
nn <-- oddget = nn := odd
END

MACHINE     Even
VARIABLES   even
INVARIANT   even : NAT & even mod 2 = 0
INITIALISATION  even := 0
OPERATIONS
evenput(nn) = PRE nn : NAT & nn mod 2 = 0
              THEN even := nn END;
nn <-- evenget = nn := even
END
```

Figure 9 The machines *Odd* and *Even*

Notice that *oddget* and *evenget* are partial events, which can be completed with an output value in each case. The *oddput*.n and *evenput*.n events are complete events corresponding to the input of a value.

These two machines are controlled by *OddCtrl* and *EvenCtrl* respectively, where

$$OddCtrl = oddget?x \rightarrow oddpass!x \rightarrow evenpass?y \rightarrow oddput!(y+1) \rightarrow OddCtrl$$
$$EvenCtrl = oddpass?z \rightarrow evenput!(z+1) \rightarrow evenpass!(z+1) \rightarrow EvenCtrl$$

The machine *Odd* will accept and maintain only odd numbers, and *Even* will accept and maintain only even numbers.

Now if we consider $OddCtrl \parallel Odd$ in isolation, we see that *OddCtrl* accepts any value y along channel *evenpass*, and then provides $y+1$ as input to *oddput*. Checking consistency will reveal that if y is odd, then *oddput* will be called outside its precondition, indicating that *OddCtrl* is not an appropriate controller for *Odd*.

However, we can see that the context of $OddCtrl \parallel Odd$, i.e., the rest of the system $EvenCtrl \parallel Even$, will ensure that the value provided for y will always be even. In fact, *OddCtrl* is a suitable controller for *Odd* in such a context.

We can include information about the guarantees provided by the context as guards on the input channels. In this case, we know that y will always be even, so we include this as a guard, adjusting *OddCtrl* to *OddCtrl*2. There are similar requirements on the input to *EvenCtrl* (in this case, that the input z is odd), so we also include a suitable guard in *EvenCtrl*:

$$OddCtrl2 = oddget?x \rightarrow oddpass!x \rightarrow evenpass?y\langle even(y)\rangle$$

$$\begin{aligned} &\rightarrow oddput!(y+1) \rightarrow OddCtrl2 \\ EvenCtrl2 = oddpass?z\langle odd(z)\rangle &\rightarrow evenput!(z+1) \\ &\rightarrow evenpass!(z+1) \rightarrow EvenCtrl2 \end{aligned}$$

However, introducing the guards themselves is not sufficient — it is necessary to establish that the context of each controller really does ensure that the guards introduced on the input channels are met. This is expressed by including the guard conditions as assumptions at the points in the context controllers where they are provided as input. This addition results in the following controllers:

$$\begin{aligned} OddCtrl3 = oddget?x &\rightarrow oddpass!x\{odd(x)\} \rightarrow evenpass?y\langle even(y)\rangle \\ &\rightarrow oddput!(y+1) \rightarrow OddCtrl3 \\ EvenCtrl3 = oddpass?z\langle odd(z)\rangle &\rightarrow evenput!(z+1) \\ &\rightarrow evenpass!(z+1)\{even(z+1)\} \rightarrow EvenCtrl3 \end{aligned}$$

It is now possible to prove (using the standard control loop invariant technique) that $OddCtrl3 \parallel Odd$ is divergence-free. This establishes that it will only ever provide odd outputs on *oddpass*, and will always correctly invoke its operations, provided it only ever accepts even numbers along *evenpass*.

Similarly, divergence-freedom of $EvenCtrl3 \parallel Even$ ensures that even numbers will only ever be passed along *evenpass* provided only odd numbers are accepted along *oddpass*. Each controller provides the correct context for the other.

The guards and assumptions were introduced to enable compositional verification: each controlled component can now be verified individually, with the necessary contextual information included in the controller description. When the components are combined, the guards and assumptions will have played their role in the verification and can be dropped. Thus we will be able to establish that

$$(OddCtrl3 \parallel Odd) \parallel (EvenCtrl3 \parallel Even) = (OddCtrl \parallel Odd) \parallel (EvenCtrl \parallel Even)$$

The technical justification for dropping the assumptions and guards is given by Theorem 9 below. Establishing this theorem is our next concern.

7.2 Manipulating guards and assumptions

Definition 5. *For a controller P, we say that predicate $E(x)$ is a* uniform guard *for input channel c if every appearance of c in P is of the form $c?x\langle E(x)\rangle \rightarrow P'$, with $E(x)$ as the guard.*

For a controller P, we say that predicate $E(v)$ is a uniform assumption *for output channel c if every appearance of c in P is of the form $c!v\{E(v)\} \rightarrow P'$ with $E(v)$ as the assumption.*

Definition 6. *We define the following translations on controller descriptions:*

$\mathcal{N}^a_C$ *removes (or* neutralises*) all assumptions from all channels* $c \in C$*;*
$\mathcal{N}^g_C$ *removes (or* neutralises*) all guards from all channels* $c \in C$*;*
$\mathcal{G}^a_C$ *transforms all assumptions on all channels* $c \in C$ *into* guards*.*

These translations can all be defined by structural induction over the syntax of controller descriptions in the standard way. Observe that the result of applying $\mathcal{G}^a_C$ is not a process controller (since it has guards on outputs), and hence will not be used to define a CSP controller. However, it is still a well-defined CSP process.

Example 6. The three translations on *OddCtrl*3 are as follows:

$$\begin{aligned}
\mathcal{N}^a_C(OddCtrl3) &= oddget?x \rightarrow oddpass!x \rightarrow evenpass?y\langle even(y)\rangle \\
&\qquad \rightarrow oddput!(y+1) \rightarrow OddCtrl3 \\
\mathcal{N}^g_C(OddCtrl3) &= oddget?x \rightarrow oddpass!x\{odd(x)\} \rightarrow evenpass?y \\
&\qquad \rightarrow oddput!(y+1) \rightarrow OddCtrl3 \\
\mathcal{G}^a_C(OddCtrl3) &= oddget?x \rightarrow oddpass!x\langle odd(x)\rangle \rightarrow evenpass?y\langle even(y)\rangle \\
&\qquad \rightarrow oddput!(y+1) \rightarrow OddCtrl3
\end{aligned}$$

The following two lemmas are useful in the technicalities of the proof of Theorem 9. The first states that if each channel in a set of channels C is associated with a guard and matching assumption, then dropping the guards on the channels, and transforming the assumptions into guards, does not change the overall behaviour.

Lemma 9. *Consider a family of process controllers* P_i*, and set of channels* C*, such that for each channel* $c \in C$ *there is some unique predicate* $E_c(x)$ *associated with* c *such that:*

- $E_c(x)$ *is a uniform guard on* c *for some* P_j*;*
- *every guard on* c *in any* P_j *is either* $E_c(x)$ *or true;*
- $E_c(v)$ *is a uniform assumption on* c *for some* P_k*;*
- *every assumption on* c *in any* P_k *is either* $E_c(v)$ *or true;*

Then it follows that

$$\Big|\Big|_i \mathcal{G}^a_C(P_i) = \Big|\Big|_i \mathcal{G}^a_C(\mathcal{N}^g_C(P_i))$$

Proof. Define $RUN_{c,E} = c?x\langle E(x)\rangle \rightarrow RUN_{c,E}$, with $\alpha(RUN_{c,E}) = \{|\ c\ |\}$. Then define $INV_C = \Big|\Big|\Big|_{c\in C} RUN_{c,E_c}$ where for each $c \in C$, E_c is the unique predicate

characterised in the statement of the lemma. The alphabet of INV_C is given by $\alpha(INV_C) = \{|\ C\ |\}$. Then INV_C allows only communications on channels in C which meet the corresponding E_c. Thus we have that

$$\begin{aligned}
\Big\|_i \mathcal{G}^a_C(P_i) &= (\Big\|_i P_i) \parallel INV_C \\
&= \mathcal{N}^g_C(\Big\|_i(P_i)) \parallel INV_C \\
&= (\Big\|_i \mathcal{N}^g_C(P_i)) \parallel INV_C \\
&= \mathcal{G}^a_C(\Big\|_i \mathcal{N}^g_C(P_i)) \parallel INV_C \\
&= \Big\|_i \mathcal{G}^a_C(\mathcal{N}^g_C(P_i)) \parallel INV_C \\
&= \Big\|_i \mathcal{G}^a_C(\mathcal{N}^g_C(P_i))
\end{aligned}$$

These steps are all justified by the semantics of parallel composition and of guards on channels. Essentially, the transformations are all possible because each channel c is blocked on $\neg E_c$ in the parallel combination.

The second lemma states that if each channel in a set of channels C is associated with a guard and matching assumption, and each controlled component is divergence-free, then the guards in their parallel combination can be dropped without introducing divergent behaviour.

Lemma 10. *Consider a family of controlled components $P_i \parallel M_i$, and set of channels C, such that for each channel $c \in C$ there is some unique predicate $E_c(x)$ associated with c such that:*

- *$E_c(x)$ is a uniform guard on c for some P_j;*
- *every guard on c in any P_j is either $E_c(x)$ or true;*
- *$E_c(v)$ is a uniform assumption on c for some P_k;*
- *every assumption on c in any P_k is either $E_c(v)$ or true;*
- *$P_i \parallel M_i$ is divergence-free for each i;*

Then it follows that $\Big\|_i(\mathcal{N}^g_C(P_i) \parallel M_i)$ is divergence-free.

Proof. Using INV_C as defined in the proof of Lemma 9, we have that

$$\Big\|_i(P_i \parallel M_i) = (\Big\|_i(\mathcal{N}^g_C(P_i) \parallel M_i) \parallel INV_C$$

Now $\Big\|_i(P_i \parallel M_i)$ is divergence-free, and so $(\Big\|_i(\mathcal{N}^g_C(P_i) \parallel M_i) \parallel INV_C$ is divergence-free. Recall $INV_C = \Big|\Big|\Big|_{c \in C} RUN_{c,E_c}$.

Now assume that there is a divergence tr of $(\Big\|_i(\mathcal{N}^g_C(P_i) \parallel M_i)$. We aim to obtain a contradiction.

- If $tr \in traces(INV_C)$ then tr is a divergence of $(\|_i(\mathcal{N}^g_C(P_i) \parallel M_i) \parallel INV_C$, contradicting the fact that this process is divergence-free.
- If $tr \notin traces(INV_C)$, then there is some event $c.v$ in tr such that $E_c(v)$ does not hold. Let $c_0.v_0$ be the first such event in tr. Then we can define $tr_0 \frown \langle c_0.v_0 \rangle$ to be the prefix of tr for which $tr_0 \in traces(INV_C)$ and $E_{c_0}(v_0)$ does not hold. Now $E_{c_0}(v)$ is a uniform assumption on c_0 for some P_k, so $tr_0 \frown \langle c_0.v_0 \rangle \upharpoonright \alpha(\mathcal{N}^g_C(P_k) \parallel M_k)$ is a divergence of $\mathcal{N}^g_C(P_k) \parallel M_k$. But every event $c.v$ in tr_0 has that $E_c(v)$ holds, and hence the trace is possible even in the presence of the guards in P_k (since none of the events in tr_0 are blocked by the guards). Thus $tr_0 \frown \langle c_0.v_0 \rangle \upharpoonright \alpha(P_k \parallel M_k)$ is a divergence of $P_k \parallel M_k$. But this contradicts the fact that $P_k \parallel M_k$ is divergence-free.

Hence it follows that $(\|_i(\mathcal{N}^g_C(P_i) \parallel M_i)$ is divergence-free, as required.

We also make use of a lemma following from Lemma 6, enabling assumptions to be dropped or replaced by guards as follows:

Lemma 11. *If $P \parallel M$ is divergence-free, then*

$$P \parallel M = \mathcal{G}^a_C(P) \parallel M = \mathcal{N}^a_C(P) \parallel M$$

Proof. This follows from Corollary 1, and the fact that the translations $\mathcal{G}^a_C$ and $\mathcal{N}^a_C$ both meet the condition of Lemma 6.

Finally, we obtain the following theorem, which enables matching guards and assumptions to be dropped if all the controlled components are divergence-free.

Theorem 9. *If $P_i \parallel M_i$ is divergence-free for each i, and for each channel $c \in C$ there is an associated predicate E_c such that*

- *$E_c(x)$ is a uniform guard on c for some P_j;*
- *every guard on c in any P_j is either $E_c(x)$ or true;*
- *$E_c(v)$ is a uniform assumption on c for some P_k;*
- *every assumption on c in any P_k is either $E_c(v)$ or true;*

then

$$\|_i(P_i \parallel M_i) = \|_i(\mathcal{N}^a_C(\mathcal{N}^g_C(P_i)) \parallel M_i)$$

Proof.

$$\begin{array}{ll} & \|_i(P_i \parallel M_i) \\ = & \qquad \{\text{generalised Lemma 11}\} \end{array}$$

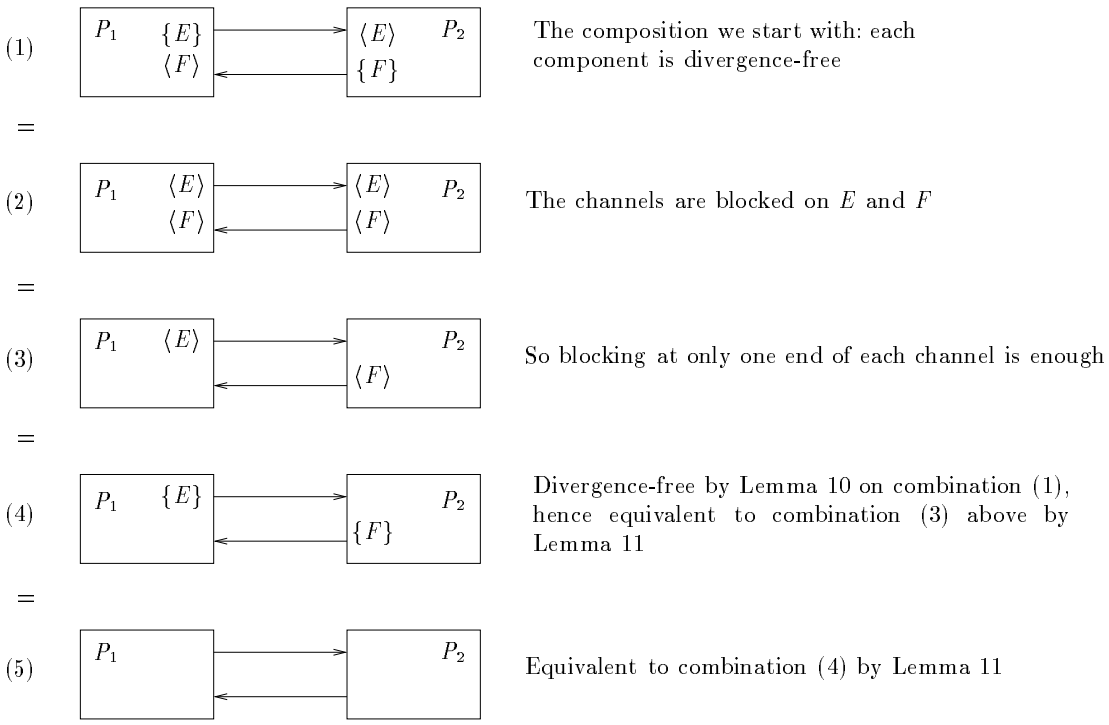

Figure 10 Illustration of the proof of Theorem 9 with two components (B machines elided)

$$
\begin{array}{ll}
 & \|_i(\mathcal{G}^a_C(P_i) \parallel M_i) \\
= & \quad \{\text{generalised Lemma 9}\} \\
 & \|_i(\mathcal{G}^a_C(\mathcal{N}^g_C(P_i)) \parallel M_i) \\
= & \quad \{\mathcal{G}^a_C \text{ distributes over parallel composition}\} \\
 & (\mathcal{G}^a_C(\|_i \mathcal{N}^g_C(P_i))) \parallel (\|_i M_i) \\
= & \quad \{\text{generalised Lemma 11,} \\
 & \qquad \text{since } (\|_i \mathcal{N}^g_C(P_i)) \parallel (\|_i M_i) \text{ divergence-free (Lemma 10)}\} \\
 & (\|_i \mathcal{N}^g_C(P_i)) \parallel (\|_i M_i) \\
= & \quad \{\text{generalised Lemma 11}\} \\
 & (\mathcal{N}^a_C(\|_i \mathcal{N}^g_C(P_i))) \parallel (\|_i M_i) \\
= & \quad \{\mathcal{N}^a_C \text{ distributes over parallel composition}\} \\
 & \|_i(\mathcal{N}^a_C(\mathcal{N}^g_C(P_i)) \parallel M_i)
\end{array}
$$

Observe that $\mathcal{N}^a_C(\mathcal{N}^g_C(P_i))$ is the process P_i with all assumptions and guards on channels in C removed. An informal picture of the proof in the case of two controlled components is given in Figure 10.

The following corollary describes the special case where each channel in C connects only two processes:

Corollary 7. *If $P_i \parallel M_i$ is divergence-free for each i, and each channel $c \in C$ is in the alphabet of exactly two of the P_i, and for each channel $c \in C$ there is an associated predicate E_c such that*

- $\diamond$ *$E_c(x)$ is a uniform guard on c for some P_j;*
- $\diamond$ *$E_c(v)$ is a uniform assumption on c for some P_k;*

then

$$\Big\|_i (P_i \parallel M_i) = \Big\|_i (\mathcal{N}^a_C(\mathcal{N}^g_C(P_i)) \parallel M_i)$$

7.3 Weakening assumptions

Theorem 9 is applicable where the assumptions and guards on a channel exactly match. In general, we require only that the assumption on a channel is stronger than the guard. This is expressed by the following theorem:

Theorem 10. *If $P_i \parallel M_i$ is divergence-free for each i, and for each channel $c \in C$ there is are two associated predicate E_c and F_c such that*

- $\diamond$ *$E_c(x)$ is a uniform guard on c for some P_j;*
- $\diamond$ *every guard on c in any P_j is either $E_c(x)$ or true;*
- $\diamond$ *$F_c(v)$ is a uniform assumption on c for some P_k;*
- $\diamond$ *every assumption on c in any P_k is either $F_c(v)$ or true;*
- $\diamond$ $\forall\, x.F_c(x) \Rightarrow E_c(x)$

then

$$\Big\|_i (P_i \parallel M_i) = \Big\|_i (\mathcal{N}^a_C(\mathcal{N}^g_C(P_i)) \parallel M_i)$$

Proof. By Lemma 6 we can replace each assumption F_c on channels $c \in C$ in each P_i by the weaker predicate E_c. In each case let the resulting process be P'_i. Then by Lemma 1 we have that

$$\Big\|_i (P_i \parallel M_i) = \Big\|_i (P'_i \parallel M_i)$$

Now the collection P'_i have matching uniform guards on the channels $c \in C$, and so Theorem 9 is applicable. Thus

$$\Big\|_i (P'_i \parallel M_i) = \Big\|_i (\mathcal{N}^a_C(\mathcal{N}^g_C(P'_i)) \parallel M_i)$$

Finally, observe that $\mathcal{N}^a_C(\mathcal{N}^g_C(P'_i)) = \mathcal{N}^a_C(\mathcal{N}^g_C(P_i))$. Thus we obtain

$$\Big\|_i (P_i \parallel M_i) = \Big\|_i (\mathcal{N}^a_C(\mathcal{N}^g_C(P_i)) \parallel M_i)$$

as required.

8 CSP State

State is captured in CSP by the use of parameters in processes which track the appropriate values. Such processes are generally defined using mutual recursion to specify how the state might be changed during execution.

For example, a process $CELL(x)$ which holds a single value x of type T for output, but which may also accept another value to hold, might be defined as follows:

$$CELL(x) = (out!x \rightarrow CELL(x)) \,\Box\, (in?y : T \rightarrow CELL(y)) \qquad (1)$$

This definition constitutes a family of definitions for a family of processes, one for each x, which are all defined in terms of each other. It may also be understood as a *vector* of process definitions indexed by T, the set of all the possible values that x can take.

In general, for a CSP semantic model S, a vector of processes X indexed by I can be thought of as a member of S^I, and is declared as follows: $X : S^I$. Then X_i is the ith element of the vector X. The vector can also be thought of as a function $I \rightarrow S$. Thus $X = \lambda\, i : I \bullet X_i$.

Then a family of functions F defining a mutually recursive set of processes is a function from one vector of processes to another: $F : S^I \rightarrow S^I$. Each F_i is a function $S^I \rightarrow S$.

For example, the $CELL$ definition above corresponds to a family of functions F indexed by T, in which each function is defined on a family of processes X also indexed by T. In this case, a particular function F_x is the function

$$F_x(X) = (out!x \rightarrow X_x) \,\Box\, (in?y : T \rightarrow X_y)$$

Thus we have that

$$F_x(CELL) = (out!x \rightarrow CELL(x)) \,\Box\, (in?y : T \rightarrow CELL(y))$$

The family of processes $CELL$ are defined to be the least fixed point of the function F. Thus for each x we have $CELL(x) = F_x(CELL)$, which matches (1) above.

Mutually recursive process definitions might involve a (finite) number of different process definitions which relate to each other, and which may have different indexing sets. For example,

$$\begin{aligned} POS(x,y) = {} & across?z \rightarrow POS(x+z, y) \\ & \Box\; up?z \rightarrow POS(x, y+z) \\ & \Box\; (\text{ if } x = 0 \text{ then } done \rightarrow TOTAL(y) \text{ else } Stop) \end{aligned}$$

$$TOTAL(y) = \begin{cases} inc \rightarrow TOTAL(y+1) & \text{if } y < 0 \\ dec \rightarrow TOTAL(y-1) & \text{if } y > 0 \\ finish \rightarrow POS(0,0) & \text{if y} = 0 \end{cases}$$

In this case POS is indexed by $\mathbb{Z} \times \mathbb{Z}$ and $TOTAL$ is indexed by $\mathbb{Z}$.

It is always possible to consider collections of indexed process definitions as a single vector of process definitions. Each definition is of the form $N_i(p_i) \mathrel{\widehat{=}} P_i$ where p_i ranges over the indexing set I_i associated with N_i. For each N_i we define a tag "N_i". The overall indexing set I is defined as follows:

$$I = \bigcup_i \{\text{“}N_i\text{”}\} \times I_i$$

and then define a single vector of definitions indexed by I as follows:

$$N(p) = P_i \text{where } p = (\text{“}N_i\text{”}, p_i) \text{ and } N_i(p_i) \mathrel{\widehat{=}} P_i$$

Thus the results we develop below are also applicable to mutually recursive collections of indexed process definitions.

8.1 Collapsing functions

Manipulation of recursively defined processes are part of the CSP folklore [Ros82,DS93]. In this paper we are concerned with the introduction and removal of state information into recursive definitions, so it will be useful to restate (and reprove) the relevant theorems here. We will construct a formal framework around the notion of *collapsing functions*.

In the following section, we use relational composition '; ' to combine mappings. Relational composition is defined as follows:

Definition 7. *If* $R_1 : S \leftrightarrow T$ *and* $R_2 : T \leftrightarrow U$ *then*

$$R_1;\ R_2 = \{(s, u) \mid (\exists\, t.(s, t) \in R_1 \wedge (t, u) \in R_2\}$$

Here R_1 is a relation between S and T; in other words, R_1 is a subset of the cartesian product $S \times T$. Similarly, R_2 is a relation between T and U.

Note that functions can also be considered as relations. Below we will also compose functions with relations and with other functions using relational composition.

Definition 8. *Given a function* $F : S^I \to S^I$ *and a set of indices* J*, a function* $c : I \to J$ *is a* collapsing function for F *if*

1. c *is surjective; and*
2. *whenever* $c(i_1) = c(i_2)$*, then, for any* $Y : S^J$*, we have* $F_{i_1}(c\ ;\ Y) = F_{i_2}(c\ ;\ Y)$.

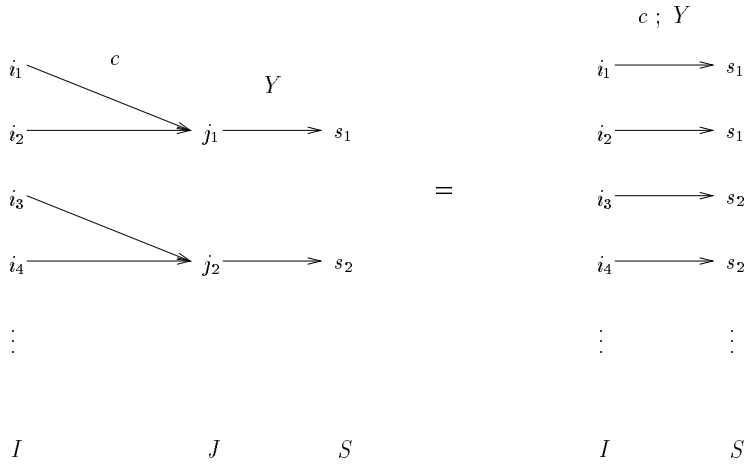

Figure 11 Transforming a vector with a collapsing function

Here, $c\ ;\ Y = \lambda\, i : I \bullet Y_{c(i)}$. Thus if $Y : S^J$ then $c\ ;\ Y : S^I$.

A collapsing function is one which identifies different components of the family of functions F. Essentially, c induces an equivalence on the set of indices I: if $c(i_1) = c(i_2)$ then i_1 and i_2 are equivalent. The function is a collapsing function if, whenever F is applied to a vector which has the same process at all equivalent indices, then the result is the same at equivalent indices. A vector Y indexed by J can be transformed to a vector indexed by I using relational composition with c, as follows: $(c\ ;\ Y) : I \to S$. In this case, equivalent indices will map to the same process. This idea is illustrated in Figure 11.

Example 7. Consider a family of functions indexed by the integers $\mathbb{Z}$ as follows:

$$\begin{aligned} F_i(Y) = {} & up \to Y_{i+1} \\ & \Box\ down \to Y_{i-1} \end{aligned}$$

This is the family of functions used in the following recursive definition:

$$\begin{aligned} MOVE(i) = {} & up \to MOVE(i+1) \\ & \Box\ down \to MOVE(i-1) \end{aligned}$$

Now we consider a singleton indexing set $J = \{0\}$. The function $c : \mathbb{Z} \to J$ defined by $c(i) = 0$ is a collapsing function. To see this, consider i_1 and i_2 such that $c(i_1) = c(i_2)$. (In fact, this is true for any $i_1, i_2 \in \mathbb{Z}$). Now consider a vector $Y : J \to S$. This will consist of a single process Y_0. For c to be a collapsing function we require that $F_{i_1}(c\ ;\ Y) = F_{i_2}(c\ ;\ Y)$. Firstly we observe that $c\ ;\ Y$ is

of type $S^{\mathbb{Z}}$ such that $(c\ ;\ Y)_i = Y_0$ for every i. In other words, it is a vector of processes in which every process is Y_0.

Now

$$\begin{aligned} F_{i_1}(c\ ;\ Y) &= (up \rightarrow (c\ ;\ Y)_{i+1}) \,\Box\, (down \rightarrow (c\ ;\ Y)_{i-1}) \\ &= (up \rightarrow Y_0) \,\Box\, (down \rightarrow Y_0) \end{aligned}$$

Similar reasoning establishes that

$$F_{i_2}(c\ ;\ Y) = (up \rightarrow Y_0) \,\Box\, (down \rightarrow Y_0)$$

which shows that c is a collapsing function.

Example 8. Consider a family of functions indexed by the natural numbers $\mathbb{N}$ as follows:

$$\begin{aligned} F_i(Y) = {} & up \rightarrow Y_{i+1} \\ & \Box\ reset \rightarrow Y_0 \\ & \Box\ (\ \mathsf{if}\ (i = 0)\ \mathsf{then}\ shutdown \rightarrow Stop\ \mathsf{else}\ Stop) \end{aligned}$$

This is the family of functions used in the following recursive definition:

$$\begin{aligned} COUNT(i) = {} & up \rightarrow COUNT(i+1) \\ & \Box\ reset \rightarrow COUNT(0) \\ & \Box\ (\ \mathsf{if}\ (i = 0)\ \mathsf{then}\ shutdown \rightarrow Stop\ \mathsf{else}\ Stop) \end{aligned}$$

The following function $c : \mathbb{Z} \rightarrow \{0, 1\}$ is a collapsing function for F:

$$c(i) = \begin{cases} 0 \text{ if } i = 0 \\ 1 \text{ if } i > 0 \end{cases}$$

To see this, consider a vector Y indexed by $\{0, 1\}$, and consider i_1 and i_2 such that $c(i_1) = c(i_2)$.

$$\begin{aligned} F_{i_1}(c\ ;\ Y) = {} & up \rightarrow (c\ ;\ Y)_{i_1+1} \\ & \Box\ reset \rightarrow (c\ ;\ Y)_0 \\ & \Box\ (\ \mathsf{if}\ (i_1 = 0)\ \mathsf{then}\ shutdown \rightarrow Stop\ \mathsf{else}\ Stop) \\ = {} & up \rightarrow Y_{c(i_1+1)} \\ & \Box\ reset \rightarrow Y_{c(0)} \\ & \Box\ (\ \mathsf{if}\ (i_1 = 0)\ \mathsf{then}\ shutdown \rightarrow Stop\ \mathsf{else}\ Stop) \\ = {} & up \rightarrow Y_1 \\ & \Box\ reset \rightarrow Y_0 \\ & \Box\ (\ \mathsf{if}\ (i_2 = 0)\ \mathsf{then}\ shutdown \rightarrow Stop\ \mathsf{else}\ Stop) \\ = {} & F_{i_2}(c\ ;\ Y) \end{aligned}$$

The penultimate line follows because $i_1 = 0$ if and only if $i_2 = 0$.

8.2 Reducing the state

The following theorem allows recursively defined families of processes to be collapsed to equivalent forms.

Theorem 11. *Let c be a collapsing function for a vector of functions F with a unique fixed point. There is some function $d : J \to I$ such that $d \,;\, c$ is the identity function on J. Let d be such a function, and define $G : S^J \to S^J$ componentwise as follows:*

$$G_j(Y) = F_{d(j)}(c \,;\, Y)$$

Then it follows that:

$\mu\, F = c \,;\, \mu\, G$

Proof. We can make the following observations for a collapsing function c:

- ◇ The choice of d makes no difference to the definition of G.
- ◇ For any $Y : J \to S$, the vector $c \,;\, d \,;\, F(c \,;\, Y) = F(c \,;\, Y)$

Let $\mu\, G$ be the least fixed point of G. Then

$$\mu\, G = G(\mu\, G) = d \,;\, F(c \,;\, \mu\, G)$$

and so

$$\begin{aligned} c \,;\, \mu\, G &= c \,;\, d \,;\, F(c \,;\, \mu\, G) \\ &= F(c \,;\, \mu\, G) \end{aligned}$$

And hence $c \,;\, \mu\, G$ is the unique fixed point of F, establishing the theorem.

Example 9. In Example 8, we have a recursively defined infinite set of processes

$$\begin{aligned} COUNT(i) = {} & up \to COUNT(i+1) \\ & \Box\ reset \to COUNT(0) \\ & \Box\ (\ \mathsf{if}\ (i = 0)\ \mathsf{then}\ shutdown \to Stop\ \mathsf{else}\ Stop) \end{aligned}$$

defined as the fixed point of the family of functions

$$\begin{aligned} F_i(Y) = {} & up \to Y_{i+1} \\ & \Box\ reset \to Y_0 \\ & \Box\ (\ \mathsf{if}\ (i = 0)\ \mathsf{then}\ shutdown \to Stop\ \mathsf{else}\ Stop) \end{aligned}$$

The vector F has a collapsing function

$$c(i) = \begin{cases} 0 \text{ if } i = 0 \\ 1 \text{ if } i > 0 \end{cases}$$

The vector G, defined by

$$G(Y) = c^{-1} \; ; F(c \; ; Y)$$

is as follows:

$$\begin{aligned} G_i(Y) = \; & up \rightarrow Y_1 \\ & \Box \; reset \rightarrow Y_0 \\ & \Box \; (\; \mathsf{if} \; (i = 0) \; \mathsf{then} \; shutdown \rightarrow Stop \; \mathsf{else} \; Stop) \end{aligned}$$

which corresponds to the following recursive definition of just two processes:

$$\begin{aligned} NEWCOUNT(i) = \; & up \rightarrow NEWCOUNT(1) \\ & \Box \; reset \rightarrow NEWCOUNT(0) \\ & \Box \; (\; \mathsf{if} \; (i = 0) \; \mathsf{then} \; shutdown \rightarrow Stop \; \mathsf{else} \; Stop) \end{aligned}$$

Theorem 11 yields that $COUNT(i) = NEWCOUNT(c(i))$ for all i, and so we obtain that $COUNT(0) = NEWCOUNT(0)$, and $COUNT(i) = NEWCOUNT(1)$ for any $i > 0$.

Theorem 11 means that if the definition of a recursive process is independent of one of the parameters in its definition, then that parameter can be dropped from the definition without affecting the behaviour of the process.

For example, the behaviour of the process $MOVE(i)$ of Example 7 is independent of the value of i. This means that this parameter can be removed from the definition of $MOVE$ without affecting its behaviour. In other words, each $MOVE(i)$ process is equivalent to the process

$$MOVE = (up \rightarrow MOVE) \; \Box \; (down \rightarrow MOVE) \tag{2}$$

Formally, this is justified by Theorem 11 with the collapsing function given in Example 7, which yields that

$$MOVE(i) = M(c(i))$$

where $c(i) = 0$, and M is defined by $M(0) = (up \rightarrow M(0)) \; \Box \; (down \rightarrow M(0))$. The single index of M is redundant, and $M(0)$ is equivalent to the version of $MOVE$ given in Line (2).

Theorem 11 also justifies the collapse of the process family $LiftCtrl4(f)$ from Section 3 to $LiftCtrl$.

Conversely, state parameters can be introduced into a recursive definition without affecting the behaviour of the process. This is achieved by introducing parameters j to a family of processes $P(i)$ in such a way that the resulting $P(i,j)$ can be collapsed to the original $P(i)$. For example, the parameter f can be introduced into *LiftCtrl* to obtain $LiftCtrl4(f)$. This can then be used as a basis for further transformations.

9 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.

This paper has obtained further results for this framework. 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 which justify the use of particular kinds of assertions.

This paper has also provided 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].

The toy examples and the case studies carried out to date have provided some experience in the way in which state, and conditions on it, are introduced into the CSP controllers. The necessary state emerges during the verification process in response to FDR checks that fail. Often it is some part of the B state that is simply duplicated in the CSP (as in our toy lift example) in order to enable verification. However, it is too early to identify patterns that may arise in this process (let alone automate it), and more case studies are being pursued.

Scalability of the approach is also a significant issue. Compositionality is a key ingredient of scalability, and it will be important to continue to identify ways in which both requirements and components can each be decomposed to minimise the amount of state required in each verification. This is the subject of ongoing research. In particular, the verification of a controlled component $P \parallel M$ against a collection of requirements might require different state to be introduced into P for each requirement, as was found in the Bounded Retransmission Protocol case study [EST03]. This is better than including all the required state for all of the required properties at once, which could result in duplicating all of the B state in the CSP controller.

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 Evans and to the anonymous reviewers for their useful comments.

References

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

[DS93] J. W. Davies and S. A. Schneider. Recursion induction for real-time processes. *Formal Aspects of Computing*, 5(6):530–553, 1993.

[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.

[Ros82] A. W. Roscoe. *A Mathematical Theory of Communicating Processes*. D. Phil thesis, Oxford University, 1982.

[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.

[ST02] 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.