Chunks: Component Verification in CSPB

Steve Schneider Helen Treharne
Department of Computing, University of Surrey

Neil Evans
School of Electronics and Computer Science, University of Southampton

S.Schneider@surrey.ac.uk, Tel: +44 1483 689637, fax: +44 1483 685061,
H.Treharne@surrey.ac.uk
ne01@ecs.soton.ac.uk

Abstract. CSP‖B is an approach to combining the process algebra CSP with the formal development method B, enabling the formal description of systems involving both event-oriented and state-oriented aspects of behaviour. The approach provides architectures which enable the application of CSP verification tools and B verification tools to the appropriate parts of the overall description. Previous work has considered how large descriptions can be verified using coarse grained component parts. This paper presents a generalisation of that work so that CSP‖B descriptions can be decomposed into finer grained components, *chunks*, which focus on demonstrating the absence of particular divergent behaviour separately. The theory underpinning *chunks* is applicable not only to CSP‖B specification but to CSP specifications. This makes it an attractive technique to decomposing large systems for analysing with FDR.

Keywords: Component based verification, B-Method, CSP, decomposition.

1 Introduction

We begin with a synopsis of the CSP‖B approach which has been under development for a number of years. The main feature of the approach is the separation of event and state based descriptions using the process algebra CSP [9] and the formal development method B [1]. One primary goal of the approach is to show that a combined specification is divergence-free [12–14]. This property is at the core of the approach, and once it has been established other safety and liveness properties of the system can be described and shown to be valid [6, 7, 15].

In [14] we discussed how a B machine can have a CSP failures-divergences semantics. We also established that a component, i.e., a parallel combination of a controller/machine pair, can be shown to be divergence-free. This means that the B operations are always called within their preconditions. In this work the CSP processes (*controllers*) themselves were divergent-free because we restricted the language to a sequential non-divergent subset of CSP.

In [12] we began to refer to collections of components as a family of processes P_i indexed by some indexing set I. Using the fact that both CSP processes and B machines have a process semantics we presented results which established the divergence freedom of a parallel combination $\|_{i \in I} P_i$. While this might be true if the whole combination $\|_{i \in I} P_i$ is considered at once, large systems will make the application of tool support to demonstrate this property impractical. Thus, we needed ways of considering parts of the system at a time, and combining the results.

In [13] we introduced a rely-guarantee style of specification which enabled us to capture *assertions*, which are predicates on values being passed on channels. We made clear that divergence freedom of the whole system depended on the divergent behaviour of individual components being prevented by the rest of the system. We discussed how to break up the system into component parts and augment each part with assertions to establish its divergence freedom. This was sufficient to deduce divergence freedom of the whole system. In that work our terminology was different. Since [6] we have settled on referring to assertions in two ways. *Blocking assertions*, previously known as guards, capture the notion of what must be true when communicating values along channels. If a blocking assertion is false communication is not permitted. With *diverging assertions*, previously known as assumptions, communication is always permitted. However, if the assertion does not hold diverging behaviour is exhibited.

The results in [13] were restrictive because the components were broken down into controller/machine pairs and one large component comprising of all the CSP processes. Furthermore, that approach does not easily scale up, since the CSP part of the system has to be checked in its entirety, and this will be subject to the limitations of CSP model-checkers.

The results presented in this paper aim to further extend our ability to perform component verification. We have coined the term *chunks* because it captures the notion of splitting the combined specification into constituent parts, i.e. a small collection of processes. A chunk can comprise of a CSP process and a B machine, or a collection of CSP processes (not necessarily all of them). We will see that a chunk gives us a much more flexible component part to deal with when aiming to establish divergence freedom.

In addition to being an extension to our integrated CSP‖B approach the results presented in this paper are applicable more generally too. They could be of potential benefit to CSP specifications that cannot be shown to be divergence-free due to tool limitations. We prove that if all the identified chunks individually do not give rise to particular divergences, then we can deduce that the whole system is divergence-free. In fact the paper is structured in two main parts and we first focus on describing how to split a collection of CSP processes into constituent parts and establishing divergence freedom to give us general results. Then the contribution to CSP‖B is presented when we demonstrate the applicability the general theory to CSP‖B. A smaller and a larger example are used to motivate the work and illustrate the results being presented.

1.1 Notation

Before presenting the first example we identify some preliminary notation. For a detailed introduction to CSP operators and the failures-divergences semantic model used in this paper the reader is referred to [11]. Similarly, the reader is referred to [1] for an overview of the B Method. The most important aspect of B to understand for this paper is that B operations are associated with preconditions, and if called outside their preconditions then they diverge. Furthermore, we do not re-iterate all the details of the syntax of CSP and B used in our approach, these can be found in [13]. Instead we discuss various points of interest when presenting the second example in Section 4.

The following introduces the key notation used in this paper, including some newly coined definitions.

The universal set of events is denoted Σ. Traces are finite sequences of events. We write $tr' \leqslant tr$ to denote that tr' is a prefix of tr. Given a set of traces D, the upwards closure of D is denoted $D\uparrow$, and is defined as follows:

$$D\uparrow = \{tr \frown tr' \mid tr \in D \wedge tr' \in \Sigma^*\}$$

Given a trace tr, its projection to a set A is denoted $tr \upharpoonright A$—this is the sequence of events from tr which are in the set A. For example,

$$\langle coin, choc, refill, coin\rangle \upharpoonright \{coin, choc\} = \langle coin, choc, coin\rangle$$

Given a set of traces D, its projection to a set A of events is defined as follows:

$$D \upharpoonright A = \{tr \upharpoonright A \mid tr \in D\}$$

CSP semantics associates with a process P a set of traces $\mathcal{T}[\![P]\!]$, a set of failures $\mathcal{F}[\![P]\!]$, and a set of divergences $\mathcal{D}[\![P]\!]$. This paper will be most concerned with the divergences.

The following lemma allows us to identify the point on a divergent trace where the process diverges.

Lemma 1. *For any non-empty divergence $tr \in \mathcal{D}[\![P]\!]$ there is a unique event a such that $\exists\, tr_0 \leq tr . tr_0 \notin \mathcal{D}[\![P]\!] \wedge tr_0 \frown \langle a\rangle \in \mathcal{D}[\![P]\!]$. In other words, there is a unique event a on which divergence was introduced in this trace.*

Proof. Let tr_0 be the maximal prefix $tr_0 \leq tr$ such that $tr_0 \notin \mathcal{D}[\![P]\!]$. There is such a tr_0, since $\langle\rangle \notin \mathcal{D}[\![P]\!]$. $tr_0 \neq tr$, since $tr \in \mathcal{D}[\![P]\!]$. Thus there is some unique a such that $tr_0 \frown \langle a\rangle \leq tr \wedge tr_0 \frown \langle a\rangle \in \mathcal{D}[\![P]\!] \wedge tr_0 \notin \mathcal{D}[\![P]\!]$.

The above lemma means that the following is well-defined:

Definition 1. *Given a process P and a divergence $tr \neq \langle\rangle, tr \in \mathcal{D}[\![P]\!]$, the* diverging event *$de(tr, P)$ is the unique event on which divergence is introduced in tr.*

Definition 2. *The set of* diverging events $de(P)$ *of a process* P *is the set* $\{de(tr, P) \mid tr \in \mathcal{D}[\![P]\!] \wedge tr \neq \langle\rangle\}$.

The *minimal divergences* $\mathcal{MD}[\![P]\!]$ of a process P is the set of divergences of P that are minimal in the prefix order.

Definition 3.

$$\mathcal{MD}[\![P]\!] = \{tr \in \mathcal{D}[\![P]\!] \mid \forall\, tr' \in \mathcal{D}[\![P]\!].tr' \leqslant tr \Rightarrow tr' = tr\}$$

Various properties can be established concerning minimal divergences. One that we will use frequently is the following:

Lemma 2. *If* $tr \in \mathcal{MD}[\![P \parallel Q]\!]$ *then* $tr \upharpoonright \alpha P \in \mathcal{MD}[\![P]\!] \vee tr \upharpoonright \alpha Q \in \mathcal{MD}[\![Q]\!]$.

A minimal divergence of $P \parallel Q$ arises from a minimal divergence of one of its components.

2 Establishing divergence-freedom

Our primary aim is to establish that a system is divergence-free. So we will consider the collection of components as a family of processes P_i indexed by some indexing set I. Thus, we aim to establish that the parallel combination $\|_{i \in I} P_i$ is divergence-free. While this might be true if the whole combination $\|_{i \in I} P_i$ is considered at once it may not be possible to check the entire system using FDR in one go. Thus, we need ways of considering parts of the system at a time, and combining the results. In general, the divergence-free operation of one part of the system might be dependent on the particular behaviour of another part.

2.1 Toy example

To illustrate the above point, we consider a small toy example in the CSP tradition: a vending machine, based on the coursework examples associated with [9]. The machine consists of a number of components, which are concerned with dispensing chocolates in return for accepting coins, refilling the machine with chocolates, and ensuring that the machine does not try to dispense chocolates when it is empty. This gives us several processes that we can group into chunks and demonstrate that they are individually divergence-free. The combination of components is illustrated in Figure 1.

The machine can diverge if too many coins are inserted without the removal of a chocolate:

$$\begin{aligned} \alpha(VM) &= \{coin, choc\} \\ VM &= coin \rightarrow ((choc \rightarrow VM) \,\Box\, (coin \rightarrow \bot)) \end{aligned}$$

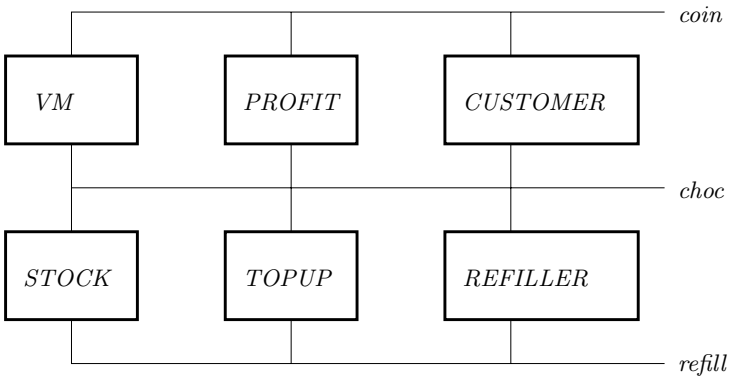

Fig. 1. Components of a vending machine

Each component has the natural alphabet: those events that are mentioned in its definition.

The profit counter can diverge if more chocolates are dispensed than have been paid for:

$$\begin{aligned} \alpha(PROFIT(i)) &= \{coin, choc\} \\ PROFIT(0) &= (coin \rightarrow PROFIT(1)) \,\Box\, (choc \rightarrow \bot) \\ PROFIT(n+1) &= (coin \rightarrow PROFIT(n+2)) \,\Box\, (choc \rightarrow PROFIT(n)) \end{aligned}$$

Refilling the machine tops it up to 17 chocolates. The stock control tracks the remaining chocolates, and can diverge if it runs out of stock:

$$\begin{aligned} \alpha(STOCK(i)) &= \{refill, choc\} \\ STOCK(0) &= (refill \rightarrow STOCK(17)) \,\Box\, (choc \rightarrow \bot) \\ STOCK(n+1) &= (refill \rightarrow STOCK(17)) \,\Box\, (choc \rightarrow STOCK(n)) \end{aligned}$$

The machine should not be refilled when it is full:

$$\begin{aligned} \alpha(TOPUP) &= \{refill, choc\} \\ TOPUP &= (refill \rightarrow FULL) \,\Box\, (choc \rightarrow TOPUP) \\ FULL &= (refill \rightarrow \bot) \,\Box\, (choc \rightarrow TOPUP) \end{aligned}$$

Finally, a counter is in control of refilling the machine:

$$\begin{aligned} \alpha(REFILLER(i)) &= \{refill, choc\} \\ REFILLER(0) &= refill \rightarrow REFILLER(17) \\ REFILLER(n+1) &= choc \rightarrow REFILLER(n) \end{aligned}$$

Finally, we include a description of the customer, who behaves well with respect to the insertion of coins:

$$\begin{aligned} \alpha(CUSTOMER) &= \{coin, choc\} \\ CUSTOMER &= coin \rightarrow choc \rightarrow CUSTOMER \end{aligned}$$

The customer will never insert more than one coin for a chocolate.

Then the overall system we wish to consider is:

$$SYSTEM = \\ CUSTOMER \parallel VM \parallel PROFIT(0) \parallel STOCK(0) \parallel TOPUP \parallel REFILLER(0)$$

We aim to establish divergence-freedom of this system. However, there are a number of points where individual components might diverge, so we will need to check that these cannot arise in the context of the overall system. Furthermore, for a larger system it may not be possible to model and check the entire system in FDR in one go.

The way we proceed is to identify for each component P the set of events that it can diverge on. We then find appropriate associated processes $Q_1 \dots Q_n$ which ensure that such divergences cannot occur. We call the subset of components $P, Q_1, \dots, Q_n$ a *chunk*, which is adequate to establish that P does not diverge on events in $de(P)$.

There are four processes which contain diverging behaviour:

- VM can diverge on some occurrences of the event *coin*. In other words, using Definition 3, any minimal divergence of VM has *coin* as its last event. If we consider $VM \parallel CUSTOMER$ we find that the resulting combination $VM \parallel CUSTOMER$ cannot diverge on *coin*. Thus $\{VM, CUSTOMER\}$ is a divergence-free chunk.
- $PROFIT(0)$ can diverge on the event *choc*. However, the parallel combination $PROFIT(0) \parallel VM$ cannot diverge on *choc*. Thus $\{PROFIT(0), VM\}$ is the appropriate chunk for this case. Note it can diverge on *coin* but this is handled by the chunk which focuses on VM with *coin* as its diverging event.
- $STOCK(0)$ can diverge on the event *choc*. However, the parallel combination $STOCK(0) \parallel REFILLER(0)$ cannot diverge on *choc*.
- $TOPUP$ can diverge on the event *refill*. However, $TOPUP \parallel REFILLER(0)$ cannot diverge on *refill*.

These are all of the cases of processes diverging on particular events. In the example all the sets of diverging events are singleton sets. However, this need not be the case in general. In each case above, there are some other processes of the system which ensure that such divergences do not occur in the system overall.

Each chunk only needs to include the process containing the diverging events and the processes which provide the control to remove those possible divergences. A chunk dealing with P need not contain all the processes that are involved in the synchronisation of the events in $de(P)$—only those that prevent the divergence from occurring are necessary. Otherwise a chunk may end up being the whole system, which is clearly not what we want because we would not have achieved any decomposition.

Furthermore, any particular diverging event can be in the set of diverging events of more than one process. In the example above, $PROFIT(0)$ and

$STOCK(0)$ can both diverge on *choc*. They are dealt with separately, each with a different chunk.

As we shall see, this is enough to guarantee that the combination of all the components, $SYSTEM$, is divergence-free.

3 Underlying theory for chunks

In a parallel system of processes, we aim to establish the property of divergence-freedom for the whole system by considering each of the processes in turn. The principal idea is to establish, for each process P, that it cannot diverge in the context of the rest of the system. Compositionality arises from the fact that we will only need to consider the relevant context, and do not need to include the parts of the system that are irrelevant to the analysis of P.

3.1 Divergences from a process

The first theorems are very general, and are concerned with the way parallel composition treats divergence. Notationally in these theorems we are thinking of P as the process which is the focus of the analysis, Q as the part of the system which ensures that P does not diverge, and R as the rest of the system. Thus we are ultimately concerned with deducing properties of the system $P \parallel Q \parallel R$ simply by considering $P \parallel Q$.

Our first lemma is concerned with the special case where $P \parallel Q$ is divergence-free. We begin with this special case because it motivates the use of minimal divergences, and gives a broad idea of the approach, which is reflected in the more general theorems.

This special case is subsumed by Lemma 4, which is itself a special case of Lemma 5. Lemma 5 supports the main theorem of this section, Theorem 1, which states when divergence-freedom of a concurrent system can be deduced from particular properties of chunks. The lemmas subsequent to Theorem 1 provide different ways of establishing those properties. In general, a variety of ways of checking chunks are possible, and an analysis will make use of different approaches for different chunks.

In Lemma 3, we can deduce that any minimal divergence of the whole system is not a minimal divergence of P. In other words, the divergence was not a result of P diverging.

Lemma 3. *If $P \parallel Q$ is divergence-free, then*

$$\mathcal{MD}[\![P \parallel Q \parallel R]\!] \upharpoonright \alpha P \cap \mathcal{MD}[\![P]\!] = \varnothing$$

Proof. Assume for a contradiction that $tr \in \mathcal{MD}[\![P \parallel Q \parallel R]\!] \upharpoonright \alpha P \cap \mathcal{MD}[\![P]\!]$. Then there is some tr_0 such that $tr = tr_0 \upharpoonright \alpha P$. Then $tr_0 \upharpoonright \alpha Q \in \mathcal{T}[\![Q]\!]$, and we already have that $tr_0 \upharpoonright \alpha P \in \mathcal{D}[\![P]\!]$. Thus $tr_0 \upharpoonright (\alpha P \cup \alpha Q) \in \mathcal{D}[\![P \parallel Q]\!]$, contradicting the fact that $P \parallel Q$ is divergence-free.

Interestingly, the theorem applies even in the case where R can diverge on events in the alphabet of P. Informally, this is because Q prevents P's divergences from occurring, and R cannot reintroduce them because of the blocking nature of the parallel operator. Of course R can introduce other divergences with the same diverging event, but it cannot introduce any divergence of P.

For example, consider the vending machine from Section 2.1 again, with VM for P, $CUSTOMER$ for Q, and define a process $BROKEN = coin \rightarrow \bot$ for R. In this case the minimal divergence of the parallel combination $P \parallel Q \parallel R$ would be $\langle coin \rangle$ but this is not be a minimal divergence of P, so the intersection would indeed be empty. The fact that $BROKEN$ diverges on $coin$ does not affect the argument that VM's divergence on $coin$ cannot occur.

Of course, in checking the whole system there would be the separate problem of finding a chunk for $BROKEN$. In fact, this will not be possible, since there is no process that prevents $coin$ from occurring. But what we have established with the chunk for VM is that if a divergence of the system does occur, it is not the fault of VM.

Observe also that Lemma 3 must be stated in terms of minimal divergences: it is not true for general divergence sets. The example above has $\langle coin, coin \rangle$ as a divergence of both VM and $VM \parallel CUSTOMER \parallel BROKEN$, even though we have that $VM \parallel CUSTOMER$ is divergence-free. The point is that the divergence $\langle coin, coin \rangle$ is not the first point at which the system diverges, so divergence of the system is the fault of $BROKEN$, on the divergence $\langle coin \rangle$. The intention of the lemma is to allow us to deduce that VM is not responsible for any divergences of $VM \parallel CUSTOMER \parallel BROKEN$, and to establish this we need to consider the minimal divergences.

The next theorem considers a more general situation, when $P \parallel Q$ possibly contains some divergences, but not as a result of P.

Lemma 4. *If* $\mathcal{MD}[\![P \parallel Q]\!] \upharpoonright \alpha P \cap \mathcal{MD}[\![P]\!] = \varnothing$, *then*

$$\mathcal{MD}[\![P \parallel Q \parallel R]\!] \upharpoonright \alpha P \cap \mathcal{MD}[\![P]\!] = \varnothing$$

Proof. Assume for a contradiction that $tr \in \mathcal{MD}[\![P \parallel Q \parallel R]\!] \upharpoonright \alpha P \cap \mathcal{MD}[\![P]\!]$. Then there is some $tr_0 \in \mathcal{MD}[\![P \parallel Q \parallel R]\!]$ such that $tr = tr_0 \upharpoonright \alpha P$. Then $tr_1 = tr_0 \upharpoonright (\alpha P \cup \alpha Q) \in \mathcal{D}[\![P \parallel Q]\!]$. Furthermore, no prefix of tr_1 is a divergence of $P \parallel Q$, since tr_0 is a minimal divergence, so $tr_1 \in \mathcal{MD}[\![P \parallel Q]\!]$. Thus $tr_1 \upharpoonright \alpha P \in \mathcal{MD}[\![P \parallel Q]\!] \upharpoonright \alpha P$, i.e. $tr \in \mathcal{MD}[\![P \parallel Q]\!] \upharpoonright \alpha P$, contradicting the fact that $tr \in \mathcal{MD}[\![P]\!]$.

This lemma would be applicable in the case of the $VM \parallel PROFIT(0)$ chunk from the previous section.

We now obtain a generalisation of Lemma 4, which is concerned only with divergence of a process P on a particular set of events A. We define $\mathcal{MD}[\![P]\!]_A$ to be those minimal divergences ending in some event in A:

$$\mathcal{MD}[\![P]\!]_A = \{tr \in \mathcal{MD}[\![P]\!] \mid last(tr) \in A\}$$

Lemma 5. *If* $\mathcal{MD}[\![P \parallel Q]\!] \upharpoonright \alpha P \cap \mathcal{MD}[\![P]\!]_A = \varnothing$*, then*

$\mathcal{MD}[\![P \parallel Q \parallel R]\!] \upharpoonright \alpha P \cap \mathcal{MD}[\![P]\!]_A = \varnothing$

Proof. Similar to the proof of Lemma 4.

This lemma is useful because in practice we may need to consider different diverging events independently of each other, and using different chunks. This will particularly arise when considering diverging assertions in process descriptions: we may wish to consider them separately. We will see an example of this in Section 4.

The main theorem, concerning the use of chunks to establish divergence freedom of a concurrent system, is the following:

Theorem 1. *Given a family of processes* $\{P_i\}_{i \in I}$*: if for each* P_i *and each* $a \in de(P_i)$ *there is some subset of processes* $\{P_j\}_{j \in J}$ *such that*

1. $P_i \notin \{P_j\}_{j \in J}$ *(i.e.* $i \notin J$*)*
2. $a \notin de(P_i \parallel \big|\big|_{j \in J} P_j)$

then $\big|\big|_{i \in I} P_i$ *is divergence-free.*

Proof. Assume for a contradiction that $tr \in \mathcal{MD}[\![\big|\big|_{i \in I} P_i]\!]$. Then there is some i such that $tr \upharpoonright \alpha P_i \in \mathcal{MD}[\![P_i]\!]$. Let $a = last(tr \upharpoonright \alpha P_i) \in de(P_i)$. Then we have a set of processes P_j for $j \in J$ meeting the two conditions above. Set

$$\begin{aligned} P &= P_i \\ Q &= \big|\big|_{j \in J} P_j \\ R &= \big|\big|_{k \neq i, k \notin J} P_k \end{aligned}$$

Then $\mathcal{MD}[\![P \parallel Q]\!] \upharpoonright \alpha P \cap \mathcal{MD}[\![P]\!]_{\{a\}} = \varnothing$, since $a \notin de(P \parallel Q)$. Then by Lemma 5 we obtain $\mathcal{MD}[\![P \parallel Q \parallel R]\!] \upharpoonright \alpha P \cap \mathcal{MD}[\![P]\!]_{\{a\}} = \varnothing$. But $tr \upharpoonright \alpha P$ is in this intersection, yielding a contradiction. Hence $\big|\big|_{i \in I} P_i$ has no divergences.

3.2 Blocking divergences

We now give some theorems which use blocking on some divergences, and divergence-freedom of the result, to establish the conditions required to apply Lemmas 4 and 5 above. These theorems will enable us to use divergence-freedom checks in FDR to obtain the results we require about the minimal divergences of processes.

If a parallel combination $P \parallel Q$, when blocked on particular minimal divergences D_P and D_Q of P and Q respectively, is divergence-free, then any minimal divergence of $P \parallel Q$ must be from one of the blocked divergences, D_P or D_Q.

Lemma 6. *If*

– $D_P \subseteq \mathcal{MD}[\![P]\!]$
– $D_Q \subseteq \mathcal{MD}[\![Q]\!]$
– $P \parallel BLOCK_{\alpha P}(D_P) \parallel Q \parallel BLOCK_{\alpha Q}(D_Q)$ *is divergence-free*

then if $tr \in \mathcal{MD}[\![P \parallel Q]\!]$, *then* $tr \upharpoonright \alpha P \in D_P$ *or* $tr \upharpoonright \alpha Q \in D_Q$.

Proof. Consider $tr \in \mathcal{MD}[\![P \parallel Q]\!]$. Assume that $tr \upharpoonright \alpha P \notin D_P$, and that $tr \upharpoonright \alpha Q \notin D_Q$. We¡ aim to establish a contradiction.

Now from Lemma 2 either $tr \upharpoonright \alpha P \in \mathcal{MD}[\![P]\!]$ or $tr \upharpoonright \alpha Q \in \mathcal{MD}[\![Q]\!]$. Assume without loss of generality that $tr \upharpoonright \alpha P \in \mathcal{MD}[\![P]\!]$.

No strict prefix of $tr \upharpoonright \alpha P$ is in D_P, since otherwise tr is not a minimal divergence of $P \parallel Q$. Thus $tr \upharpoonright \alpha P \in \mathcal{MD}[\![(P \parallel BLOCK_{\alpha P}(D_P))]\!]$, since $BLOCK_{\alpha P}(D_P))]$ does not restrict $tr \upharpoonright \alpha P$.

Similarly, no strict prefix of $tr \upharpoonright \alpha Q$ is in D_Q, since otherwise tr is not a minimal divergence. Thus $tr \upharpoonright \alpha Q \in \mathcal{T}[\![(Q \parallel BLOCK_{\alpha Q}(D_Q))]\!]$. Hence we obtain that $tr \in \mathcal{MD}[\![P \parallel BLOCK_{\alpha P}(D_P) \parallel Q \parallel BLOCK_{\alpha Q}(D_Q)]\!]$, yielding a contradiction.

The following corollary of Lemma 6 will be useful in establishing condition (2) for Theorem 1, since it provides a way of establishing the antecedent in particular cases. It is concerned with divergence on particular events from the set A. It states that if divergence-freedom can be obtained by blocking some minimal divergences not ending in A, then no minimal divergence of $P \parallel Q$ arises from a divergence of P ending in A.

Corollary 1. *If*

1. $D_P \subseteq \mathcal{MD}[\![P]\!] \setminus \mathcal{MD}[\![P]\!]_A$
2. $D_Q \subseteq \mathcal{MD}[\![Q]\!] \setminus \mathcal{MD}[\![Q]\!]_A$
3. $P \parallel BLOCK_{\alpha P}(D_P) \parallel Q \parallel BLOCK_{\alpha Q}(D_Q)$ *is divergence-free*

then $\mathcal{MD}[\![P \parallel Q]\!] \upharpoonright \alpha P \cap \mathcal{MD}[\![P]\!]_A = \varnothing$

The first condition means that D_P cannot contain any traces ending in events from A; and the second condition means that D_Q cannot contain any traces ending in events from A. Lemma 6 yields that any $tr \in \mathcal{MD}[\![P \parallel Q]\!]$ either has $tr \upharpoonright \alpha P \in D_P$, or else $tr \upharpoonright \alpha Q \in D_Q$, and in both cases $last(tr) \notin A$, so we obtain $tr \upharpoonright \alpha P \in \mathcal{MD}[\![P]\!]_A$.

3.3 Assertions

When we are working with CSP descriptions, we will often aim to decompose them by introducing assertions onto the channels. This is especially true for CSP‖B, the driving motivation for this work. We will aim to establish that the assertions are always true within system executions. This will be achieved by considering them as diverging assertions (i.e. the process diverges if the assertion is false), and establishing divergence-freedom of the resulting system.

This is exactly the kind of situation that Theorem 1 is intended to apply to. That theorem allows consideration of divergences separately, so we can consider diverging assertions on different channels separately. In order to do this, we can convert other assertions (those that are not the focus of interest) to blocking assertions. The following theorems give the justification for this, and relate this syntactic transformation to the results of the preceding section.

Lemma 7. *If Q' is Q with diverging assertions on channels in a set C replaced by blocking assertions, then $Q' = Q \parallel BLOCK_{\alpha Q}(D)$ for some set of divergences $D \subseteq \mathcal{MD}[\![Q]\!]_{\{|C|\}}$.*

Proof. By structural induction over Q. This includes the case for parallel, thus Q can be a parallel combination of processes.

The following corollary shows how we can use a divergence-freedom check of $P' \parallel Q'$, which is $P \parallel Q$ with suitable diverging assertions replaced by blocking assertions, to show that certain divergences of P are not divergences of $P \parallel Q$.

Corollary 2. *If c is a channel of the process P such that*

1. *P' is the process P with some diverging assertions replaced by blocking assertions, but with no assertions on c replaced*
2. *$\{|\ c\ |\} \cap de(Q) = \varnothing$*
3. *Q' is the process Q with some diverging assertions replaced by blocking assertions.*
4. *$P' \parallel Q'$ is divergence-free*

then $tr \in \mathcal{MD}[\![P \parallel Q]\!] \upharpoonright \alpha P \cap \mathcal{MD}[\![P]\!]_{\{|c|\}} = \varnothing$

This is a restatement of Corollary 1 in the light of Lemma 7.

It can also be written in a form which more directly corresponds to condition (2) of Theorem 1, showing how that condition can be established for particular process channels.

Corollary 3. *Given a family of processes $\{P_i\}_{i \in I}$, and a particular P_i and subset of processes $\{P_j\}_{j \in J}$, if c is a channel of the process P_i such that*

1. *P_i' is the process P_i with some diverging assertions replaced by blocking assertions, but with no assertions on c replaced*
2. *For each $j \in J$, $\{|\ c\ |\} \cap de(P_j) = \varnothing$*
3. *For each $j \in J$, P_j' is the process P_j with some diverging assertions replaced by blocking assertions.*
4. *$P_i \parallel \left|\left|\right|_{j \in J} P_j'$ is divergence-free*

then for any $a \in \{|\ c\ |\}$ we have $a \notin de(P_i \parallel \left|\left|\right|_{j \in J} P_j)$.

3.4 Chunks with B machines

The motivation for this work arises from the need to consider CSP processes in parallel with B machines. B machines have a CSP failures-divergences semantics, so the theorems from Section 3.3 are applicable, and B machines are considered as CSP processes from the point of view of Theorem 1. However, they are not written using CSP syntax, so the syntax-oriented approach from Section 3.3 of manipulating assertions is not applicable. However, some chunks may require the consideration of some B machines in a system description, since it may be the constraints imposed by the B machine that prevent divergence in some other component. In order to make use of B machines in chunks within Theorem 1, we need another approach.

There is already a theory for establishing divergence-freedom of $P \parallel M$, where M is a B machine and P is a CSP controller whose alphabet contains that of M. M interacts with P via its *machine channels*, where channel inputs and outputs correspond to operation calls of the B machine. Divergence-freedom is established by identifying a *control loop invariant* CLI which is true every recursive call of P, and which ensures that divergence does not arise on an individual traversal of the control. Further details of this approach can be found in [14].

The following lemma will be useful in establishing condition (2) of Theorem 1 in cases where the chunk contains a B machine. This will be necessary in cases where the process under consideration (i.e. the P_i) is the B machine.

Lemma 8. *If C is the set of machine channels of machine M, and P is the controller for M, and P' is the process P except that non-machine channels block in P' where they diverge in P, and $P' \parallel M$ is divergence-free, then any event in $\{|\ C\ |\}$ is not a diverging event for $P \parallel M$.*

Proof. This is justified by Corollary 1. If D is P's set of non-machine channels, then we are blocking P on $\mathcal{MD}[\![P]\!]_{\{|D|\}}$, and M on $D_M = \varnothing$. Then we have the following conditions:

1. $D_M \subseteq \mathcal{MD}[\![M]\!] \setminus \mathcal{MD}[\![M]\!]_{\{|C|\}}$
2. $D_P \subseteq \mathcal{MD}[\![P]\!] \setminus \mathcal{MD}[\![P]\!]_{\{|C|\}}$
3. $M \parallel BLOCK_{\alpha M}(D_M) \parallel P \parallel BLOCK_{\alpha P}(D_P)$ is divergence-free

The first follows because $D_M = \varnothing$; the second follows because only minimal divergences on channels in D are blocked; and the third we have from the fact that $M \parallel P'$ is divergence-free. Thus we can deduce from Corollary 1 that $\mathcal{MD}[\![M \parallel P]\!] \upharpoonright \alpha M \cap \mathcal{MD}[\![M]\!] = \varnothing$.

This lemma is useful, since we can use CLI techniques to show that $P' \parallel M$ is divergence-free.

3.5 Pulling it all together

The overall aim is to provide support for establishing that a CSP‖B system description is divergence-free. Divergences can arise when B operations are called

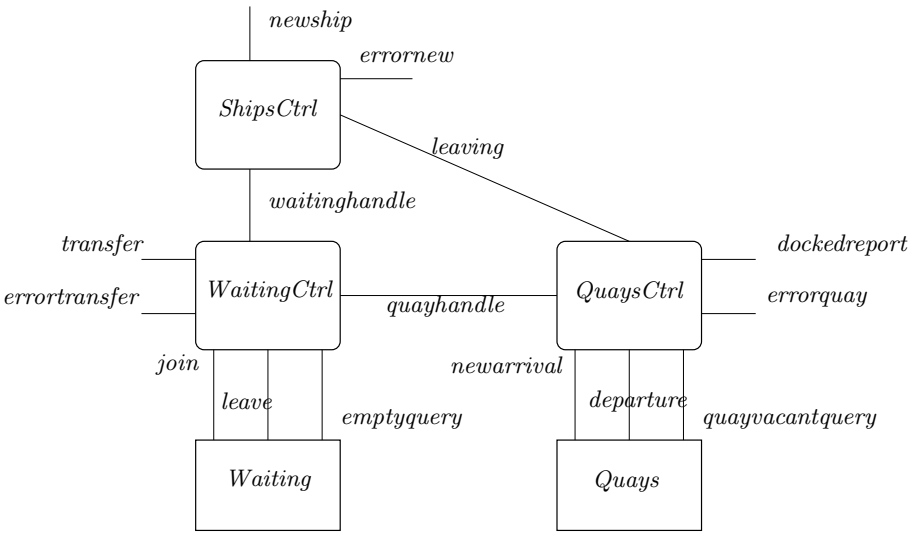

Fig. 2. Port system architecture

outside their preconditions, and it is the responsibility of the designers of the CSP controllers to ensure that this does not occur.

The B machines must be considered as chunks with their controllers providing their environment. In many situations, it may be the case that a CSP controller cannot guarantee this in isolation for the machine under its control, and that constraints imposed by the rest of the system also need to be factored in. Existing results allow divergence-freedom to be established for a machine in parallel with a single sequential process, and so information from other parallel components needs to be introduced into the controller as assertions on the channels.

Having introduced diverging assertions onto some of the channels within the CSP parts of the system, we then need to identify chunks to verify that these diverging assertions do not introduce new divergences. For chunks consisting purely of CSP processes, the divergence-freedom check can be carried out using FDR. The next section will illustrate this process.

4 Example

We will begin by describing the structure of the system and its components. Then we will show how to verify that the system is divergence-free using the *chunks* technique. Consider the following system:

$$PortSystem = ShipsCtrl \parallel WaitingCtrl \parallel QuayCtrl \parallel Waiting \parallel Quays$$

It comprises of three controllers and two machines as shown in Figure 2. It tracks the entering of ships and their docking at quays. A ship can enter a port

MACHINE *Waiting*
SEES *Bool_TYPE* , *Context*
VARIABLES *waiting*
INVARIANT *waiting* ∈ iseq (*SHIP*)
INITIALISATION *waiting* := [ ]
OPERATIONS
join (*ss*) ≙ **PRE** *ss* ∈ *SHIP* ∧ *ss* ∉ ran (*waiting*)
THEN *waiting* := *waiting* ← *ss*
END ;
ss ⟵ **leave** ≙ **PRE** *waiting* ≠ [ ]
THEN *ss* := first (*waiting*) ‖ *waiting* := tail (*waiting*)
END ;
bb ⟵ **emptyquery** ≙
IF *waiting* = [ ]
THEN *bb* := *TRUE*
ELSE *bb* := *FALSE*
END
END

Fig. 3. *Waiting* machine

and proceed to a waiting queue. A ship can be transferred from the waiting queue to a quay. A ship can also leave a quay so that it is no longer tracked by the system. Only one ship is allowed to be docked at a particular quay. A ship cannot be both waiting and docked at a quay at the same time.

The main state associated with this example is the waiting queue, and the quays at which ships can dock. This information is captured as an injective sequence of ships, *waiting*, and a partial injective function, *docked*. We would naturally use B to hold this data in separate machines and provide a rich set of operations to manipulate it. Figure 3 defines the *Waiting* machine which contains three operations: `join`, `leave` and `emptyquery`. The first operation appends a new ship to the waiting queue. The second operation extracts a ship from the head of the waiting queue. The third operation does not change the state of the queue and simply examines whether the waiting queue is empty.

Figure 4 defines the `Quay` machine which also contains three operations: `newarrival`, `departure` and `queryvacantquay`. The first operation allocates a ship to a randomly unoccupied quay. The second operation removes the ship which is docked at a particular quay *qq*. The third operation examines whether the quay *qq* is not occupied by a ship. In both machines we note that the types for ships and quays are given in a separate context machine, `Context`, which is accessible by all machines (but not provided here).

Let us now consider how these operations are called within their associated controllers. Our initial attempt at defining controllers is given in Figure 5 and their aim is to co-ordinate the main behaviour of ships we identified earlier. In addition to *WaitingCtrl* and *QuaysCtrl* we have included a *ShipsCtrl* to illus-

```
MACHINE Quays
SEES Bool_TYPE , Context
VARIABLES docked
INVARIANT docked ∈ QUAY ⤖ SHIP
INITIALISATION docked := ∅
OPERATIONS
  quay ⟵ newarrival ( ss ) ≙
    PRE ss ∈ SHIP ∧ dom ( docked ) ≠ QUAY ∧ ss ∉ ran ( docked ) THEN
      ANY qq
      WHERE qq ∈ QUAY − dom ( docked )
      THEN docked ( qq ) := ss || quay := qq
      END
    END ;
  ss ⟵ departure ( qq ) ≙ PRE qq ∈ dom ( docked )
    THEN docked := { qq } ⩤ docked || ss := docked ( qq )
    END ;
  bb ⟵ quayvacantquery ( qq ) ≙ PRE qq ∈ QUAY
    THEN IF qq ∈ dom ( docked )
      THEN bb := FALSE
      ELSE bb := TRUE
      END END
END
```

Fig. 4. *Quays* machine

trate that the CSP‖B architecture can accommodate processes which have no corresponding B machines.

The *ShipsCtrl* regulates the arrival of new ships using the *newship* event and observes their departure using the *leaving* event. This process keeps track of all the ships being monitored in the port system using the set *ss*.

Once a new ship has arrived along channel *waitinghandle* the *WaitingCtrl* aims to place it appropriately in the waiting queue by interacting with the B machine along channel *join*. The *WaitingCtrl* also transfers a ship from the waiting queue so that the *QuaysCtrl* can record its docking at a quay. To achieve this transfer the *WaitingCtrl* queries the B machine, using *queryemptyquay*, to check that the waiting queue is non-empty, then a communication along *leave* extracts an appropriate ship which is passed to *QuaysCtrl* along *quayhandle*, which in turn records the docking using the operation *newarrival*.

QuayCtrl also can deal with ships vacating a particular quay. A similar pattern of behaviour is followed where the B machine is queried to check that the quay is occupied, and if it is, the appropriate ship is extracted and *QuayCtrl* communicates with the *ShipsCtrl* to record its removal from the system using the *leaving* event.

$$
\begin{aligned}
SC(ss) =& newship?s \rightarrow \textbf{if } (s \in ss) \\
& \quad \textbf{then } errornew \rightarrow SC(ss) \\
& \quad \textbf{else } waitinghandle!s \rightarrow SC(ss \cup \{s\}) \\
& \square\ leaving?s \rightarrow SC(ss - \{s\}) \\
ShipsCtrl =& SC(\{\})
\end{aligned}
$$

$$
\begin{aligned}
WaitingCtrl = & waitinghandle?s \rightarrow join!s \rightarrow WaitingCtrl \\
& \square\ transfer \rightarrow emptyquery?b \rightarrow \\
& \quad \textbf{if } (b = true) \\
& \quad \textbf{then } errortransfer \rightarrow WaitingCtrl \\
& \quad \textbf{else } leave?s \rightarrow quayshandle!s \rightarrow WaitingCtrl
\end{aligned}
$$

$$
\begin{aligned}
QuaysCtrl = & quayshandle?s \rightarrow newarrival!s?q \rightarrow dockedreport!s!q \rightarrow QuaysCtrl \\
& \square\ vacate?q \rightarrow quayvacantquery!q?b \rightarrow \\
& \quad \textbf{if } (b = true) \\
& \quad \textbf{then } errorquay \rightarrow QuaysCtrl \\
& \quad \textbf{else } departure!q?s \rightarrow leaving!s \rightarrow QuaysCtrl
\end{aligned}
$$

Fig. 5. Initial CSP Controllers

4.1 Adding assertions to the controllers

In the previous section our informal description of the various patterns of behaviour that port can undertake glossed over the fact that the four modifying operations of B have nontrivial preconditions. It is the responsibility of the controllers to make sure that the operations are called within their preconditions so that they do not contribute any divergent behaviour. Let us re-examine the *WaitingCtrl* and the *QuaysCtrl* controllers. The *ShipsCtrl* can remain as in Figure 5.

Consider the `join` and `leave` operations called by *WaitingCtrl*. The precondition of the `join` operation signifies that only a ship that is not already waiting should be considered. The control flow of the processes gives us this, and so a synchronisation with *waitinghandle* ensures that the ship is a valid one. However, our aim is to prove divergence freedom by considering controller/machine pairs as a single chunk in the following section. Consequently, any assumptions we need to make about ships in order to ensure that the precondition of `join` holds have to be recorded in the *WaitingCtrl*. We have two ways of doing this: first we can call a query operation to see whether the ship is in the waiting queue before calling `join`, second we can annotate the appropriate channel with an assertion as is shown in Figure 6. The assertion needs to capture the fact that the ship s is not already in the queue. Therefore, the CSP needs to contain some state so that this can be expressed. It need not capture all the details of the order of the queue, it only needs to record the set of ships in the queue, represented by *wss*. This is an abstraction of the B state and is duplicated in the CSP for the purposes of proof, after which it can be dropped again. The assertion is placed

$$
\begin{aligned}
WC(wss) =& (waitinghandle?s\{s \notin wss\} \rightarrow join!s \rightarrow WC(ss \cup \{wss\})) \\
& \Box\ (transfer \rightarrow emptyquery?b \rightarrow \\
& \qquad \textbf{if}\ (wss = \{\}) \\
& \qquad \textbf{then}\ errortransfer \rightarrow WC(wss) \\
& \qquad \textbf{else}\ leave?s\{s \in ss\} \rightarrow quayshandle!s \rightarrow WC(wss - \{s\})) \\
WaitingCtrl =\ & WC(\{\})
\end{aligned}
$$

$$
\begin{aligned}
QC(dss) =& (quayshandle?s\{s \notin dss\} \rightarrow newarrival!s?q \rightarrow \\
& \qquad\qquad dockedreport!s!q \rightarrow QC(dss \cup \{s\})) \\
& \Box\ (vacate?q \rightarrow quayvacantquery!q?b \rightarrow \\
& \qquad \textbf{if}\ (b = true) \\
& \qquad \textbf{then}\ errorquay \rightarrow QC(dss) \\
& \qquad \textbf{else}\ departure!q?s\{s \in dss\} \rightarrow leaving!s \rightarrow QC(dss - \{s\})) \\
QuaysCtrl =\ & QC(\{\})
\end{aligned}
$$

Fig. 6. Augmented CSP controllers

at the point in the process where the value of the ship is bound, i.e., on *waitinghandle*. We choose the second option because we know that when the whole system is composed together the control flow ensures that the precondition is true and therefore we don't need the extra overhead of calling a query operation. We only need these annotations when breaking down the proof into chunks.

The `leave` operation should only be called when the queue is non-empty. Here, we use a query operation to check whether the queue is empty. We cannot add a diverging assertion because they are checked after communication has occurred. What we require is a way of blocking the call to the operation when *wss* is empty. We cannot use a blocking assertion on the machine channel because this would violate our non-discriminating property discussed in [13]. Therefore, a query operation is the only way to provide the required level of control here.

Second, the way in which we augment the *QuaysCtrl*, so that we will be in a position to prove the chunk $QuaysCtrl \parallel Quays$, is similar to the way we augmented *WaitingCtrl*. We annotate the *quayhandle* channel with a diverging assertion to check that the ship to be docked is not already docked. Thus, when we call `newarrival` the precondition will hold. The assertion is expressed using the set *dss* in *QuaysCtrl*. The set is an abstraction of the docked ships. Here again, the CSP description does not need to be concerned about where the ships are docked, it simply needs to track whether they are or not. We also provide a query operation to provide the required level of control prior to calling the `departure` operation.

Notice that there are two further assertions in Figure 6, and both of these are as a result of a factoring in the constraints imposed by the rest of the system. We need to know that the ship being passed to the *QuaysCtrl* is in fact in the waiting queue, and the diverging assertion $\{s \in wss\}$ will enable us to check this. Then we will be sure that the ships in the waiting queue and the quays

do not overlap. Similarly, we need to know that when a ship is leaving that it was indeed one of the ships in a quay, and the diverging assertion $\{s \in dss\}$ will enable us to check this. Then when the state of *ss* in *ShipCtrl* is updated we can be sure that the ship removed from the quay will also be removed from the whole system.

4.2 Splitting the Port System into chunks

The resulting system from the above discussion is a port system comprising of three controllers, two of which contain diverging assertions; and two machines. We will now demonstrate the application of Theorem 1 on each of the processes to show that the overall *PortSystem* is divergence-free.

Consider the *Waiting* machine. An appropriate chunk is $WaitingCtrl \parallel Waiting$. in order to demonstrate that this is divergence-free we would need to show that none of the machine channels are diverging events of $WaitingCtrl \parallel Waiting$. First we transform the diverging assertion on *waitinghandle* to a blocking assertion but leave the diverging assertion on *leave*, then we can show using the CLI technique that the transformed controller $WaitingCtrl' \parallel Waiting$ is divergence-free. Applying Lemma 8 enables us to conclude that $WaitingCtrl \parallel Waiting$ is divergence free.

Consider the *Quays* machine. An appropriate chunk is $QuaysCtrl \parallel Quays$. In order to demonstrate that this is divergence-free we need to show that none of the machine channels are diverging events of $QuaysCtrl \parallel Quays$. First we transform the diverging assertion on *quayshandle* to a blocking assertion but leave the diverging assertion on *departure*, then we can show using the CLI technique that the transformed controller $QuaysCtrl' \parallel Quays$ is divergence-free. Applying Lemma 8 enables us to conclude that $QuaysCtrl \parallel Quays$ is divergence free.

Now we can turn our focus to the controllers and consider the application of Theorem 1 for each of them. There is nothing to prove for *ShipsCtrl* because it does not contain any diverging events.

Consider *WaitingCtrl*. It can diverge on two channels *waitinghandle* and *leave*. We need not identify a chunk to show the absence of divergence on *leave* because we have already done that as part of the first chunk above. Consider diverging on *waitinghandle*. We require the co-operation of *ShipsCtrl* to discharge this. We transform the *leave* assertion into a blocking assertions in line with condition 1 in Corollary 3. Then we can establish that $ShipsCtrl \parallel WaitingCtrl$ is divergence-free using FDR and this meets condition 4 of Corollary 3. Applying Lemma 8 enables us to conclude that {it waitinghandle} is not a diverging event of $WaitingCtrl \parallel Waiting$.

Consider *QuaysCtrl*. It can diverge on two channels *quayhandle* and *departure*. We need not identify a chunk to show the absence of the divergence on *departure* because we have already done that as part of the second chunk above. Consider diverging on *quayhandle*. We require the co-operation of *WaitingCtrl* to discharge this diverging assertion. First we transform the diverging assertions on *leave* from *WaitingCtrl* and *departure* from *QuayCtrl* and this is in line

with conditions 1 and 3 in Corollary 3. Then we can show using FDR that $QuaysCtrl \parallel WaitingCtrl$ is divergence-free and this meets condition 4 of Corollary 3. *ShipsCtrl* is a simple P_j since it is itself divergence-free and therefore condition 2 and 3 hold. This allows us to conclude that *quayshandle* does not contribute any divergent behaviour.

The above has shown that we can show that all chunks are divergence-free. Using results from [13] we can then remove any of the assertion annotations and this allows us to conclude that the original port system is divergence-free.

In the above example, it would have been possible to group all the controllers together and prove divergence freedom of the controllers on their own, and then prove the individual controller/machine pairs and deduce that the combination is divergence-free. This was the necessary approach in [13]. What we have shown above is that we can systematically go through each process in a combined system, and show that it is divergence free either by using the CLI technique or in FDR once we have transformed the chunk into an appropriate form. We have had some concerns over the scalability of the previous approach because it may not always be possible to group all the controllers together. This work gives us a much finer grained way of proving divergence-freedom which we feel will potentially be more scalable than the previous technique presented in [13].

5 Conclusion

Our principle of composing components and describing their interactions resonates in the *Reo* coordination model [2]. The main differences is that Reo does not concentrate on the entities that are connected to the interacting components, rather the method is concerned with governing the flow of data between components and developing support for dynamic reconfiguration of its connectors. The work that is most closely related to ours is csp2B [3] and the recent extensions to proB [4]. The former is a purely syntactic transformation of a CSP description into B. The CSP descriptions can be very expressive, allowing interleaving, but the work does not address any compositional verification issues. Very promising new work is emerging which has the same semantic foundations as our approach. The authors of [4] also consider a B machine as a process that can engage in events in the same way that a CSP process can and so an operation call is a synchronisation of a CSP event with its corresponding operation. To the best of our knowledge they have not yet examined the issues related to the scalability of their approach. Their work currently focuses on automating consistency checking of a combined CSP and B specification. One future avenue worth pursuing would be to investigate whether the ProB form of consistency checking could be tailored to automating the CLI proof that we carry out to show that a $P \parallel M$ is divergence-free.

References

1. Abrial, J.-R.: *The B-Book*. Cambridge University Press, 1996.

2. Arbab, F.: *Reo: A channel-based coordination model for component composition*. Mathematical Structures in Computer Science (14), 329–366, 2004.
3. Butler M. J.: *csp2B: A Practical Approach to Combining CSP and B*, Formal Aspects of Computing, Volume 12 (2000).
4. Butler, M. and Leuschel, M.:*Combining CSP and B for Specification and Property Verification*. In Proceedings of Formal Methods 2005 (in press), Newcastle upon Tyne. Fitzgerald, J., Hayes, I. and Tarlecki, A., Eds. 2005. Helen
5. Cavalcanti A., Sampaio A., and Woodcock J.: *Refinement of Actions in Circus*, In REFINE'02, FME Workshop, Copenhagen (2002).
6. Evans, N., Treharne, H.: *Investigating a File Transfer Protocol Using CSP and B*. SoSym Journal (accepted for publication 2005).
7. Evans N., Treharne H., Laleau R., Frappier M.: *How to Verify Dynamic Properties of Information Systems*. 2nd International Conference on Software Engineering and Formal Methods, IEEE, China, 2004.
8. Formal Systems (Europe) Ltd.: Failures-Divergences Refinement: FDR2 User Manual 1997.
9. Hoare, C. A. R.: *Communicating Sequential Processes*. Prentice Hall, Englewood Cliffs, 1985.
10. Leuschel M., Butler M.: *ProB: A Model Checker for B*. Proceedings FME 2003, Pisa, Italy, LNCS 2805, pages 855-874. Springer, 2003.
11. Schneider, S. A.: *Concurrent and Real-Time Systems: the CSP Approach*, John Wiley 1999.
12. Schneider S.,Treharne H.: *Communicating B Machines*. ZB2002, Grenoble, LNCS 2272, Springer, January (2002).
13. Schneider, S., Treharne, H.: *CSP Theorems for Communicating B Machines*. In Proceedings of IFM 2004, LNCS 2999, Springer-Verlag, University of Kent, 2004.
14. Treharne H., Schneider S.: *Using a Process Algebra to control B OPERATIONS*. In K. Araki, A. Galloway and K. Taguchi, editors, IFM'99, York (1999), pp437–456.
15. Treharne H., Schneider S., Bramble M.: *Composing Specifications using Communication*. In ZB2003, LNCS 2651, Springer-Verlag, Finland, 2003.