Composing Specifications using Communication

Helen Treharne, Steve Schneider, and Marchia Bramble

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

E-mail: {helen,steve,marchia}@cs.rhul.ac.uk

Abstract. This paper develops a case study using the process algebra CSP to enable controlled interaction between B machines. This illustrates how B machines are essential components within a combined communicating system. The development steps used to build the case study are new; they are applications of theoretical results which allow us to focus on the external interface of a combined communicating system, compositionally verify it, and show that it is a refinement of a more abstract specification described in CSP. This allows safety and liveness properties to be established for combinations of communicating B machines.

Keywords: B-Method, CSP, Composing Specifications, Combining Formalisms, Concurrency.

1 Introduction

This paper focuses on a case study which illustrates the development steps to be followed in order to achieve a verified system specification expressed using the B-Method [1] and the process algebra of Communicating Sequential Processes (CSP) [3]. The B-Method is used to specify the data-rich aspects of the system whilst CSP naturally captures the flow of events within a system. We use B and CSP since they both have mature and existing tool support for verification [6, 5]. The contribution of the paper is both the detailed development of the case study and the methodology adopted in order to achieve the final system.

The paper applies previous and recent results [10–14] in the development steps which form our methodology for developing combined communicating systems. The results define a theoretical framework for specifying combined specifications using CSP and B. Previously, we focused on proving that a collection of *controllers* (Ps) are consistent with their underlying *machines* (Ms). A single controller P is a CSP process which can encapsulate a single flow of control for a B machine M and some of its events correspond directly to B operations. Consistency between P and M is proven by identifying a *control loop invariant* (CLI): a predicate on the state of the B machine and the indexes (which correspond to process state) of the CSP controller, which must be true at any recursive call. This provides a link between the state of the B machine and the process state,

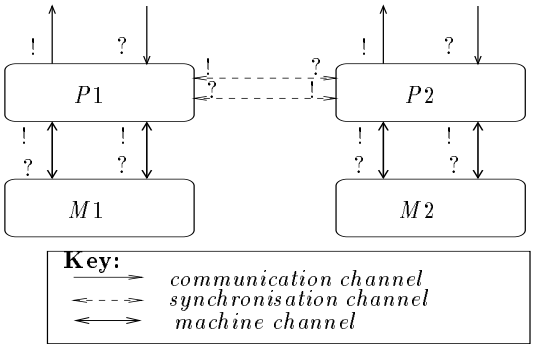

Fig. 1. An architecture for concurrent B machines

which must be strong enough to ensure non-divergence (i.e. operations must be called within their preconditions). Proof obligations to establish this were identified in [14]: essentially, a translation of the CSP control loop into corresponding AMN must be strong enough to guarantee establishment of the *CLI*.

In [10] we identified an architecture to co-ordinate interaction between collections of combinations (of B machines *Ms* and their controllers *Ps*). This architecture, is shown in Figure 1, and is again adopted in this paper so that any communication is between a machine and controller pair ($P \parallel M$), between two or more controllers, or between a controller and its external environment. Two results support composition of the components:

Result 1 If all the CSP controller/controlled B machine pairs are divergence-free, then so is the entire parallel combination. This is a useful result, since we already know how to establish that individual CSP/B pairs are divergence-free using the *CLI* technique.

Result 2 If the system is divergence-free, and the parallel combination of the CSP controllers *without their controlled B machines* is deadlock-free, then the entire parallel combination is deadlock-free. This is a valuable result, since the combination of CSP controllers can be checked using FDR, and this enables the results of such a check to be lifted to include the B components.

The architecture adopted in this paper was not in used in our early work [12, 14] because external events interacting with the system were only used to set up appropriate values for B variables modelling the environment of an embedded system, and thus we permitted their flow of control to permeate into the underlying B machines. In our current work we view a B machine as a component which interacts only with its immediate environment which is a CSP controller.

In this paper we emphasise how a B machine can be viewed as a component at the heart of a combined communicating system but where the overall system interface is defined as interactions with the external environment in the CSP description. In general, our (CSP $\parallel$ B) approach does not preclude B operations of a machine from contributing to this interface but we do not discuss this in this

paper. Having identified this interface we can encapsulate (via CSP hiding) any internal system activity. We apply recent results to show even after hiding this behaviour we can still prove that a collection of *controllers* (Ps) remain consistent with their underlying *machines* (Ms) and that the controllers themselves are divergence-free. Furthermore, we will also show that a combined communicating system ($Ps \parallel Ms$) is a refinement of a more abstract specification described solely in CSP. This allows safety and liveness properties to be established for the combined system.

We now outline developments steps we use to build combined communicating systems, so that we can refer to them throughout the paper.

Step 1 Define the individual B machines,
Step 2 Give CSP controllers for them that describe the flow of control for their use,
Step 3 Prove consistency between each of the B machines and their controllers to establish divergence freedom,
Step 4 Prove deadlock freedom of the combination of the controllers,
Step 5 Establish that hiding internal events does not introduce any further divergences,
Step 6 Verify safety and liveness properties of the system,
Step 7 Steps 3 and 4 may only be possible if we introduce appropriate guards and assumptions, and extra events into the CSP. Steps 5 and 6 may only be possible if we introduce state as necessary into the CSP.
Step 8 If state was introduced in the CSP it can be dropped following verification so that we end up with the system we had in Step 5 and 6 (but which has now been verified).

Steps 3 and 4 are applications of Results 1 and 2 above. Steps 5 and 6 can be performed independently but both need to be achieved by the end of the development of a particular system. They are also likely to have an iterative effect on the development. Similarly, Step 7 means that we have to re-do Steps 1-6. When more state is added to the CSP it is likely to be more difficult to achieve Step 3 due to the complex interaction of B machines and their controllers.

This paper is organised as follows: Section 2 introduces the CSP controller language and semantics; Section 3 introduces the recent consistency results which under-pin our CSP|| B approach; Section 4 describes the case study development in detail; and Section 5 ends with a discussion. The paper assumes familiarity with AMN; further details can be found in [1].

2 CSP Controllers and B Machines

2.1 Notation for Controllers

CSP is a language for describing processes of concurrent systems and their patterns of interactions. The unit of interaction is the atomic *event* which processes perform and on which they may synchronise. Events can be unstructured (such

as *start*), or they can have some structure, generally of the form of a channel name c and some values v that are passed along a channel. Thus the occurrence of $c.3.5$ may be understood as the passing of the values 3 and 5 along the channel c. The *occurrence* of events is atomic. The set of all events is denoted Σ.

We will use a subset of CSP to describe the controllers for B machines. The language we use is based on the language in [10, 13, 14] and is given by the following pseudo-BNF rule;

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

where a, c and d can be both *communication* and *synchronisation channels*, and e is a *machine channel*, x represents all data variables on a channel, v represents all data values being passed along a channel, $E(x)$ is a predicate on x (it may be elided, in which case it is considered to be *true*), b is a boolean expression, and p is a process expression. Note that the notation for machine channels has reverted to standard CSP notation, where '?' denotes input to the process, and '!' denotes output from the process. Previously in [10, 14] we considered CSP processes as wrappers for the B machines rather than controllers, and so the notation described communications on machine channels from the point of view of the B machine (i.e. **!** for output from the machine, and **?** for input to it). Although the semantics remains the same, the change in notation reflects the shift in our view of the CSP process.

The process $a \rightarrow P$ can perform an event a and then behave as P. The input process $c?x\langle E(x)\rangle \rightarrow P(x)$ denotes a process that can accept any input x along channel c, provided x satisfies the guard E. It will block other inputs. Having accepted x, it will behave as $P(x)$. Conversely, the output process $d!v\{E(v)\} \rightarrow P$ will initially output v along channel d. If v meets the assumption E then it will behave as P, otherwise it will diverge.

Communication on machine channels involves both input and output, though special cases might drop one or even both of these. A machine channel e will correspond to an operation $x \longleftarrow e(v)$ of the underlying B machine, matching the input v of the operation to the output from the CSP, and the output x of the operation to the CSP input. This communication can include either an assumption or a guard. The communication $e!v?x\{E(x)\} \rightarrow P(x)$ can accept any x as input, but will diverge if $E(x)$ is not satisfied. Conversely, the communication $e!v?x\langle E(x)\rangle \rightarrow P(x)$ can only accept inputs x which satisfy the guard $E(x)$. Inputs which fail the guard are blocked.

The external choice, $P_1 \,\Box\, P_2$, is initially prepared to behave either as P_1 or as P_2, with the choice being made on occurrence of the first event. The choice of the first event is made by the environment of the choice. Conversely, the choice $P_1 \sqcap P_2$ chooses internally whether to behave as P_1 or as P_2, and its environment

has no control over the way the choice is resolved. Indexed internal choice ($\sqcap$) chooses a value x such that is meets the predicate $E(x)$ and then behaves as the process P which may depend on the value of x. Another form of choice is controlled by the value of a boolean expression in an **if** expression.

$S(p)$ is a process name where p is an expression. Each process expression contains a recursive call, $S(p)$. For example, a process which manages a set of values can be described by two recursive families Set and $Flush$ indexed by sets:

$$\begin{aligned} Set(S) = {} & in?x\langle x \notin S\rangle \rightarrow Set(S \cup \{x\}) \\ & \Box\ (\sqcap_{v \in S}\ out!v \rightarrow Set(S - \{v\})) \\ & \Box\ \mathit{flush} \rightarrow Flush(S) \end{aligned}$$

$$Flush(S) = \textbf{if } S = \{\} \textbf{ then } Set(S) \textbf{ else } (\sqcap_{v \in S}\ out!v \rightarrow Flush(S - \{v\}))$$

Observe the use of the guard on the input channel in to block the input of any value which is already in the set S; and that some arbitrary member of S is selected for output. If the event flush is chosen then the behaviour is described by $Flush$, which only allows outputs until the set is empty.

2.2 Composing Controllers

In addition to the language for controllers, CSP provides a number of composition operators. These can be used to combine controller processes, and can also be applied to B machines considered as CSP processes, since they can be given a CSP semantics. The operators we are concerned about in this paper are the following:

$$P_1 \parallel P_2 \mid \Big\|_i P_i \mid P_1 \,|||\, P_2 \mid \Big|||_i P_i \mid P \setminus A$$

The *parallel composition* operator, $P_1 \parallel P_2$, executes P_1 and P_2 concurrently, requiring that they synchronise on events in both their alphabets, and allowing independent performance of events outside their alphabets. In this paper the alphabet of a process will be all the events that it can perform. This allows messages to pass along channels. There is also an indexed form $\|_i P_i$.

For example, if the processes $Copy$ and $Copy2$ are defined as follows

$$\begin{aligned} Copy &= in?x \rightarrow mid!x \rightarrow Copy \\ Copy2 &= mid?y \rightarrow out!y \rightarrow Copy2 \end{aligned}$$

then $Copy \parallel Copy2$ can input values v on in, have both components synchronise on $mid.v$ which passes the value to $Copy2$, and then have $Copy2$ independently output v on out.

A special form of parallel composition is *interleaving*, which allows concurrent processes to execute completely independently. The combination $P_1 \,|||\, P_2$ executes both P_1 and P_2 in parallel, but without any synchronisation, even on common events. For example, $Copy \,|||\, Copy$ behaves as a bag of capacity 2: it can

accept up to two messages on channel *in*, and can output them independently on channel *mid*. There is also an indexed interleaving operator $|||_i P_i$.

Finally (for this paper), the *hiding* operator $P \setminus A$ executes P with the set of events A as internal events. This is often used on parallel combinations of processes to make their communication channels internal once they have been connected together. For example, the process $(Copy \parallel Copy2) \setminus \{| mid |\}$ behaves as a two place buffer with input channel *in* and output channel *out*.

2.3 Semantics

CSP processes are identified with the observations that can be made of them: thus the semantics of a CSP process will be a set of observations. The precise form of the observations will describe the CSP model. The *traces model* uses traces as observations. The *stable failures model* uses traces along with subsequent refusals. The *failures/divergences model* uses traces, divergences, and failures. We briefly describe them here. A fuller explanation can be found in [8].

A *trace* tr of a process P is a finite sequence of events that it may be observed to engage in. The *traces model* identifies a process with its set of traces.

A *divergence* of a process P is a sequence of events tr such that P reaches a divergent state (which may be thought of as entering a non-terminating loop, or in specification terms as a specification which allows any behaviour) during the performance of the sequence of events tr. A process is *divergence-free* if it has no divergences. Divergence denotes undesirable behaviour, and it is generally useful to establish that a process is divergence-free.

A *refusal* of a process P is a set X of events that P might be initially prepared to refuse. A *stable failure* of a process P is a trace/refusal pair (tr, X) such that P can initially perform the sequence of events tr, and reach a non-divergent state in which every event in X is refused. The stable failures of P is denoted $\mathcal{F}_{SF}[\![P]\!]$.

If for some tr $(tr, \Sigma) \in \mathcal{F}_{SF}[\![P]\!]$ then P can reach a state in which no event at all is possible, and we say that P has a *deadlock*. If there is no such stable failure, then P is *deadlock-free*.

The semantic models allow *refinement*: $P \sqsubseteq Q$ means that the semantics of Q is a subset of the semantics of P. This allows a process P to be treated as a specification of allowed behaviours, and Q meets the specification if $P \sqsubseteq Q$. In this paper we use trace refinement $\sqsubseteq_T$ and stable failures refinement $\sqsubseteq_F$.

The CSP model-checker FDR allows checks for refinement between processes. It also allows checks for deadlock and divergence freedom to be made automatically for CSP processes.

B machines can also be given a semantics in any of these models [4, 14] which means that they also can be combined (with CSP processes, and with each other) using CSP operators.

3 Recent Results on Combining Communicating Systems

In this section we present how the new development steps 5-8 (outlined in Section 1) are justified as a result of recent theoretical extensions to our CSP || B approach.

3.1 Recent results

Result 2 in Section 1 shows that results about the CSP part of a combined system can be lifted to the entire combination. Recent results [11] have identified further ways in which checks on the CSP part of the combined system can provide results about the overall combination. In particular, the following results are useful. In cases (3) and (4), they are in the context of a divergence-free combination $Ps \parallel Ms$, where $Ps = \|_i P_i$ and $Ms = |||_i Mi$, so these results are applicable only after divergence-freedom has been established. (Here the Mi machines are completely independent and do not communicate directly, so they are combined using interleaving).

Result 3 If $Ps \setminus Int$ is divergence-free, then so too is $(Ps \parallel Ms) \setminus Int$. If we want to declare some channels as internal (Int), this might introduce some divergent behaviour, so it is necessary to check in any particular case that this has not occurred. This result enables this check to be carried out purely on the CSP part of the combination.

Result 4 If Ps and Ps' differ only on the divergences of Ps (i.e., any failure (tr, X) of one but not the other must have that tr is a divergence of Ps) then $Ps \parallel Ms = Ps' \parallel Ms$. In other words, the behaviour of Ps and Ps' *in the context of* Ms is the same. Informally, this is because Ms prevents Ps from reaching any of its divergent behaviour. In practice, as will be illustrated in the case-study, this result enables assumptions on CSP channels to be dropped or transformed into guards. This is because assumptions are simply predicates on values which lead to divergent behaviour if the predicate is false. If $P_i \parallel M_i$ is divergence-free then all the assumptions must be true. Replacing assumptions in P_i with the same predicate as a guard, or removing the assumption entirely, yields a process P_i' which is the same as P_i on the non-divergent behaviour of P_i. This means that the combination Ps can be transformed to a combination Ps' which is the same on the non-divergent behaviour of Ps, and which might be more suitable for checking in FDR.

Result 5 If αM is the set of all channels of the B machines, and $\alpha M \subseteq Int$, and Ps does not have any guards on any of its channels, then $Ps \setminus Int \sqsubseteq_F (Ps \parallel Ms) \setminus Int$. In other words, once all the communications with the underlying state machines are hidden (and possibly others), then to show some failures property of the system $SPEC \sqsubseteq_F (Ps \parallel Ms) \setminus Int$, it is sufficient to show that the same property holds purely for the CSP controllers Ps when those channels are hidden: $SPEC \sqsubseteq_F Ps \setminus Int$. This result relies on the fact that the B machines that we are concerned with do not block on any channel, and that the CSP processes do not partially block on any internal channel in Int: in other words, they must be

open to any input on a channel if they are open to some. A similar argument holds for trace refinement so that if we can show that a trace property holds for a restricted controller interface it can also be lifted up to the combination.

We will also make use of a folk theorem (which is also given in [11]) concerning mutual recursion:

Result 6 If the behaviour of a CSP process is completely independent of an index in its recursive definition, then that index can be dropped.

The development steps in Section 1 are applications of these results; Results 3 and 4 are related to Step 5, Result 5 is related to Step 6, and Result 6 is related to Step 8.

4 Example

The example in this section is of a bank control system which processes the flow of customers through a bank. The bank contains only one counter and a number of queues in which customers line up. They can proceed from these queues to the counter in order to process their business. Customers can enter the bank provided the bank has not reached its limit of customers. Customers can also leave the bank once they have finished their business. This example was originally inspired by the distributed buffer example described in Circus [2] where the cache and various flags were described separately from the ring.

The development is highly iterative because placing the state appropriately within the architecture to facilitate verification is not straightforward. Thus, we go through several incremental versions as follows;

Version 1 identifies all the important state and models it solely within B machines (**Step 1**). When required this state information is then passed via parameters into and retrieved from the CSP as appropriate. Having developed the controllers (**Step 2**) the consistency of the combined system (**Step 3 and 4**) is established. Then an attempt to verify the desired system properties (**Steps 5** and **6**) is made and mostly fails.

Version 2 re-structures the architecture so that more state is added to the CSP (**Step 7**) in order to restrict its behaviour sufficiently so that a better attempt can be made at verifying the desired properties. This re-structuring eliminates the need for one of the B machines, but the state in the other machine is so complicated that expressing it solely in CSP is not appropriate, and so it is an essential part of the system. As we stated in Section 1 this re-structuring means that we have to re-do **Steps 1-6**. The advantage of the resulting architecture is that it minimises the synchronous communication needed between controllers before appropriate underlying state changes can occur (since less retrieval of state occurs from the underlying B).

Version 3 differs only slightly from version 2. Nonetheless, this version is needed so that we can confidently check (using FDR) that the bank system is free from any additional divergences which could have been introduced by the hiding of internal events.

MACHINE *Counter*
SEES *Types*
VARIABLES $currentCustomer$, $queueNo$
INVARIANT $currentCustomer \in CUSTOMER \wedge queueNo \in QUEUENUM$
INITIALISATION
$currentCustomer := defaultCustomer \parallel queueNo := 1$
OPERATIONS
setCustomer ($cc \in CUSTOMER$) $\widehat{=}$ $currentCustomer := cc$;
$cc \longleftarrow$ **getCustomer** $\widehat{=}$ $cc := currentCustomer$;
setnextQueue ($qNo \in QUEUENUM$) $\widehat{=}$ $queueNo := qNo$;
$qq \longleftarrow$ **getQueueNo** $\widehat{=}$ $qq := queueNo$
END

Fig. 2. *Counter* Machine

The B-Toolkit [6] and FDR source files for the example can be downloaded[1].

4.1 Version 1 of the Bank System

In the overview of this version above we stated that all the main state of the system is captured in B machines. We separate the state into two machines so that one machine tracks all the information related to the counter and the other deals with queues and the waiting customers. These two machines are called *Counter* and *Queues*, which are defined in Figures 2 and 3 respectively.

The single bank counter is captured in the *Counter* machine. It introduces the variable *currentCustomer* to track the person currently being serviced. The set of all possible customers, *CUSTOMER* is declared in a separate context machine, *Types*, as shown in Figure 4. Similarly, the set of queue numbers, *QUEUENUM*, is declared in the same context machine. This global visibility of types is typical in B developments.

Four very simple operations are offered by *Counter*: *setCustomer* simply assigns a customer to the counter; *getCustomer* queries the current counter; *setNextQueue* updates the queue number to the next one to be serviced; and *getQueueNo* outputs the queue value.

The *Queues* machine tracks customers in the different queues within a bank by updating the *customerQueues* variable appropriately. The invariant of the machine provides constraints on the queues of customers, stating that customers should only ever appear in at most one queue, and in at most one position. It also captures the fact that there is a safety limit of *maxQueueingCustomers* representing the total number of queueing customers allowed in the bank.

Four operations are offered by the machine. The operation *joinQueue* non-deterministically adds a customer to the end of one of the shortest queues.

[1] http://www.cs.rhul.ac.uk/home/helen/papers/zb2003/sources.tar.gz

MACHINE *Queues*
SEES *Types*
VARIABLES *customerQueues*
INVARIANT $customerQueues \in QUEUENUM \rightarrow \mathsf{iseq}\ (\ CUSTOMER\ ) \wedge$
$\mathsf{card}\ (\ \mathsf{union}\ (\ \mathsf{ran}\ (\ customerQueues\ )\ )\ ) \leq maxQueueingCustomers \wedge$
$\forall\ (\ c1\ ,\ c2\ )\ .\ (\ c1 \in \mathsf{dom}\ (\ customerQueues\ ) \wedge$
$c2 \in \mathsf{dom}\ (\ customerQueues\ ) \wedge$
$c1 \neq c2 \Rightarrow \mathsf{ran}\ (\ customerQueues\ (\ c1\ )\ ) \cap \mathsf{ran}\ (\ customerQueues\ (\ c2\ )\ ) = \varnothing\ )$
INITIALISATION $customerQueues := QUEUENUM \times \{\ [\ ]\ \}$

OPERATIONS

joinQueue $(\ cc \in CUSTOMER\ ) \mathrel{\widehat{=}}$
PRE $cc \notin \mathsf{ran}\ (\ \mathsf{union}\ (\ \mathsf{ran}\ (\ customerQueues\ )\ )\ ) \wedge$
$\mathsf{card}\ (\ \mathsf{union}\ (\ \mathsf{ran}\ (\ customerQueues\ )\ )\ ) < maxQueueingCustomers$
THEN
ANY *number* **WHERE** $number \in \mathsf{dom}\ (\ customerQueues\ ) \wedge$
$\mathsf{size}\ (\ customerQueues\ (\ number\ )\ ) =$
$\mathsf{min}\ (\ \mathsf{ran}\ (\ \lambda\ xx\ .\ (\ xx \in \mathsf{dom}\ (\ customerQueues\ )\ \mid$
$\mathsf{size}\ (\ customerQueues\ (\ xx\ )\ )\ )\ )\ )$
THEN
$customerQueues\ (\ number\ ) := customerQueues\ (\ number\ ) \leftarrow cc$
END
END ;
$cc \longleftarrow$ **leaveQueue** $(\ queueNo \in QUEUENUM\ ) \mathrel{\widehat{=}}$
PRE $customerQueues\ (\ queueNo\ ) \neq [\ ]$ **THEN**
$cc := \mathsf{first}\ (\ customerQueues\ (\ queueNo\ )\ )\ \|$
$customerQueues\ (\ queueNo\ ) := \mathsf{tail}\ (\ customerQueues\ (\ queueNo\ )\ )$
END ;
$bb \longleftarrow$ **queryQueueEmpty** $(\ queueNo \in QUEUENUM\ ) \mathrel{\widehat{=}}$
IF $customerQueues\ (\ queueNo\ ) = [\ ]$ **THEN**
$bb := yes$
ELSE
$bb := no$
END ;
$bb \longleftarrow$ **queryIsInQueue** $(\ cc \in CUSTOMER\ ) \mathrel{\widehat{=}}$
IF $cc \in \mathsf{ran}\ (\ \mathsf{union}\ (\ \mathsf{ran}\ (\ customerQueues\ )\ )\ )$ **THEN**
$bb := yes$
ELSE
$bb := no$
END
END

Fig. 3. *Queues* Machine

MACHINE *Types*
SETS $CUSTOMER$; $QSTATUS = \{ yes , no \}$
CONSTANTS $defaultCustomer$, $maxQueueingCustomers$, $numQueues$
PROPERTIES $defaultCustomer \in CUSTOMER \wedge maxQueueingCustomers \in \mathbb{N} \wedge$
$numQueues = \mathsf{card}\ (\ QUEUENUM\ )$
DEFINITIONS $QUEUENUM \ \widehat{=}\ \mathbb{N}_1$
END

Fig. 4. *Types* Machine

The precondition needs to state that the customer is not already in some other queue and that the $maxQueueingCustomers$ constraint has not been reached. The operation *leaveQueue* removes the customer from the head of a particular queue and then all the other customers can move along. Neither updating operation is robust. Therefore, the CSP controllers will have to protect the flow of control appropriately. In order to assist with this we provide two query operations. The operation *queryQueueEmpty* reports whether a particular queue of customers is empty or not. Note that if the output is *no* we know that it is safe to call the *leaveQueue* operation. We also provide the operation *queryIsInQueue* to determine whether a customer is already in a queue. This will be useful when it comes to ensuring that the precondition of the *joinQueue* operation is met. We need not provide a query operation for the cardinality constraint because there will be enough information in the CSP controllers to discharge this.

Now let us consider the controllers which drive the *Counter* and *Queues* machines so that the operations of these B machines are called appropriately within the overall system. We define one controller for each machine, *CounterCtrl* and *QueuesCtrl*, and so our combined system is $BankSystem = ((CounterCtrl \parallel Counter) \parallel (QueuesCtrl \parallel Queues))$. Figure 5 illustrates the overall architecture of the whole system and highlights all the channels involved. The main external interface of the system is given by the *enterBank*, *report*, and *leaveBank* communication channels.

CounterCtrl is a controller which deals with customer requests, and is given in Figure 6. This is the process which takes overall control of communicating with the environment. It is defined in terms of a parameterised recursion. The parameter *num* represents the number of people in the bank (i.e. the customer at the counter and all the ones in queues). Initially, there are no such customers and so *num* is set to zero. There are two main execution paths, described using subprocesses, which are controlled by an external choice. The first path processes customers entering the bank and the second allows customers to leave the bank.

Let us first look at the process, *JoinCtrl*, which deals with customers entering the bank. It is not always possible to allow customers to enter a bank, since there is a maximum limit on the number of people in the bank, represented by the constant $maxLimit$. If this limit is not reached and a customer enters

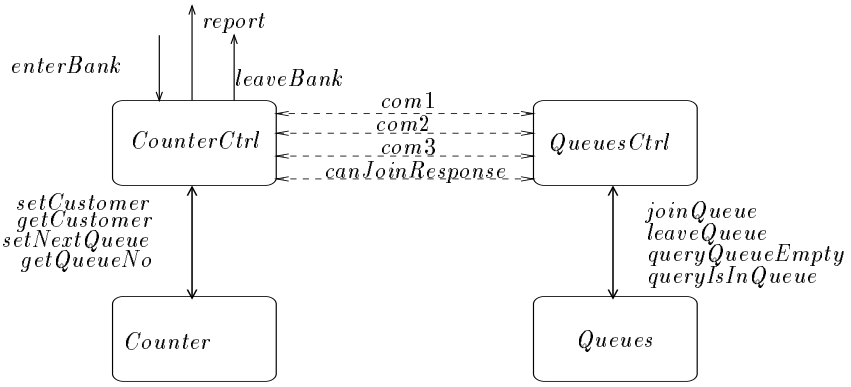

Fig. 5. Communication Architecture of *BankSystem*- Version 1

the bank there are two further branches. First, if there are no customers at all (i.e. when *num* is zero) the customer can proceed straight to the counter and so the event *setCustomer* is performed, and in turn this corresponds to setting the B variable *currentCustomer* to *cc*. Second, if there are customers present in the bank we must check that the new *cc* passed along the communication channel *enterBank* is not the one at the counter. If it is then we simply ignore the request to allow *cc* to enter for a second time, and we report that the request was unsuccessful with a *fail* communication along the channel *report*. However, if *cc* is a different customer to the one at the counter we can send a synchronisation communication (to the *QueuesCtrl* described below) to indicate that *cc* should join an appropriate queue.

Now let us consider the process, *LeaveCtrl*, which deals with leaving customers and there are two main possible behaviours. First, if there is only one customer then the customer to leave the bank is the one associated with *cc* (i.e. the one at the counter). This information is retrieved using the *getCustomer* event. Then a communication along *leaveBank* indicates *cc*'s departure, after which we update the current customer to a default value and reset the number of people in the bank to zero. Second, if there is more than one customer in the bank, then the person to leave is again the one at the counter but in this case we must also move a person from a queue to the counter to be serviced next. We do this by querying which queue is to be serviced using *getQueueNo* and send this information along *com*2 so that the *QueueCtrl* can extract the appropriate customer. We then wait for this information to be received along *com*3 along with the new queue number (which is the next queue to be serviced next time round). Following this synchronised communication we can perform the setting events to update the counter and queue information. The process ends with a recursive call in which the total number of people in the bank is decremented by one.

$$
\begin{array}{l}
CounterCtrl \quad = CurrentCtrl(0) \\
CurrentCtrl(num) = JoinCtrl(num) \,\Box\, LeaveCtrl(num) \\
JoinCtrl(num) = \\
\quad num < maxLimit \,\&\, enterBank?cc \rightarrow \\
\quad ((num = 0 \,\&\, report.success \rightarrow setCustomer.cc \rightarrow CurrentCtrl(num+1)) \\
\quad \Box \\
\quad (num > 0 \,\&\, getCustomer?currentCust \rightarrow \\
\qquad \textbf{if } (cc = currentCust) \\
\qquad \textbf{then} \\
\qquad\quad report.fail \rightarrow CurrentCtrl(num) \\
\qquad \textbf{else} \\
\qquad\quad com1!cc \rightarrow canJoinResponse?bb \rightarrow report.bb \rightarrow \\
\qquad\quad (\textbf{if } (bb = success) \\
\qquad\quad \textbf{then} \\
\qquad\qquad CurrentCtrl(num+1) \\
\qquad\quad \textbf{else} \\
\qquad\qquad CurrentCtrl(num)) \quad)) \\
\\
LeaveCtrl(num) = \\
\quad (num = 1 \,\&\, getCustomer?cc \rightarrow leaveBank!cc \rightarrow \\
\quad setCustomer.defaultCustomer \rightarrow CurrentCtrl(0)) \\
\quad \Box \\
\quad (num > 1 \,\&\, getCustomer?cc \rightarrow leaveBank!cc \rightarrow \\
\quad getQueueNo?qNo \rightarrow com2!qNo \rightarrow com3?newCust?newQNo \rightarrow \\
\quad setCustomer!newCust \rightarrow setNextQueue!newQNo \rightarrow CurrentCtrl(num-1))
\end{array}
$$

Fig. 6. *Counter* Controller

The other main controller is *QueuesCtrl* which drives the *Queues* machine to process the queueing customers in the bank. *QueuesCtrl* is defined in Figure 7, in terms of a parameterised mutual recursion. The parameter s holds the maximum number of queueing customers (one less that the bank limit). Initially, there are no such customers.

There are two main possible execution paths in *QueuesCtrl*; the first deals with joining a queue, and the second processes leaving a queue. The first branch receives a communication requesting that *cc* joins a queue. As we stated earlier the operation *joinQueue* is not robust and so this is reflected in the CSP by a query event and then wrapping an if statement around the main *joinQueue* event (which effectively changes the underlying B state). If the customer *cc* is already in another queue the request to join fails and the appropriate communication is sent back to the *CounterCtrl*, otherwise it can proceed and the number of people s can be incremented.

The second path synchronises on the receipt of a queue number. Subsequently, a call to the *NextQCtrl* sub-process occurs. The main event in this pro-

$QueuesCtrl = QCtrl(0)$
$QCtrl(s) =$
$(s < maxQueueingCustomers \,\&\, com1?cc \rightarrow queryIsInQueue!cc?bb \rightarrow$
if $(bb = yes)$
then
$canJoinResponse!fail \rightarrow QCtrl(s)$
else
$canJoinResponse!success \rightarrow joinQueue.cc \rightarrow QCtrl(s+1))$
$\Box$
$(s > 0 \,\&\, com2?qNo \rightarrow NextQCtrl(s, qNo))$
$NextQCtrl(s, queueNo) =$
$queryQueueEmpty!queueNo?bb \rightarrow$
if $(bb = no)$
then
$leaveQueue!queueNo?cc \rightarrow com3!cc!inc(queueNo) \rightarrow QCtrl(s-1)$
else
$NextQCtrl(s, inc(queueNo))$

Fig. 7. *Queues* Controller

cess is the one which effectively removes a customer from the queue, *leaveQueue*, but again the corresponding B precondition needs to hold. Therefore, we use a query operation and an if statement in a similar way to the previous branch. From this we can clearly see that the underlying B machine has a direct impact on the style of specification of a controller when we have complete information hiding and fragile operations.

If there is a queue of customers associated with the queue number *qNo* we can then extract a customer and communicate this to the *CounterCtrl* along *com*3. We also pass the new queue information back to the other controller. Finally, we make a recursive call to *QCtrl* so that we are prepared to deal with further queueing related requests.

The *NextQCtrl* controller is itself an iterative process because if *qNo* is associated with an empty queue the controller needs to cycle through the queues in the bank until a non-empty queue is found.

The above completes our discussion of **Steps 1** and **2** for this first version of the bank system. Now we turn to the verification steps; **Step 3** requries that we verify $CounterCtrl \parallel Counter$ and $QueuesCtrl \parallel Queues$ to be divergence-free. The *CLI* for $CounterCtrl \parallel Counter$ is simply *true* because we do not have to ensure any preconditions are met other than typing ones. An appropriate *CLI* for $QueuesCtrl \parallel Queues$ is

$$s = \Sigma_{i \in dom(customerQueues)} card(customerQueues(i))$$

and again we can show that this holds true. Informally, this is because each time we add a customer to a queue we increment s by one and the cardinality of one of the queues will also increase by one. Similarly, when we remove a person

$$SPEC = |||_{i \in \{1..maxLimit\}} CUST$$
$$CUST = enterBank?i \rightarrow (report.fail \rightarrow CUST \;\Box\; report.success \rightarrow leaveBank.i \rightarrow CUST)$$

$$SPEC2 = NEWSPEC(0)$$
$$NEWSPEC(num) = num < maxLimit \,\&\, enterBank?cc \rightarrow (report.success \rightarrow NEWSPEC(num+1) \sqcap report.fail \rightarrow NEWSPEC(num)) \;\Box\; (num > 0 \,\&\, \prod_{cc \in CUSTOMER} leaveBank.cc \rightarrow NEWSPEC(num-1))$$

Fig. 8. Desired Properties of *BankSystem*

V	*LivelockFree*	$SPEC \sqsubseteq_T V$	$SPEC2 \sqsubseteq_T V$	$SPEC2 \sqsubseteq_F V$
1	No	No	Yes	No
2	No	Yes	Yes	Yes
3	Yes	Yes	Yes	Yes

Fig. 9. Verification of *BankSystemControllers* in FDR

from the queue we decrement s and this is reflected by the removal of a customer from a queue. Establishing deadlock-freedom of the *BankSystem* (**Step 4**) simply involves checking that $BankSystemControllers = (CounterCtrl \parallel QueuesCtrl)$ is deadlock-free (which is the case).

The remaining checks to be performed are for livelock-freedom (**Step 5**) and the verification of safety and liveness properties (**Step 6**). We cannot prove livelock-freedom because a *yes* result can always be passed along the *queryQueueEmpty* channel and thus we loop round the *NextQCtrl* process infinitely.

A desired safety property of our system is one where a customer entering the bank can also leave at some point later and customers are allowed to use the bank independently. An appropriate liveness property will guarantee that whenever possible (given that there is a limit on the number of people allowed in the bank at any one time) the system should always offer the possibility of letting a customer enter or leave the bank. Figure 8 captures these properties as CSP processes. We need to show that the *BankSystemControllers* meet these specification using FDR and thus we can show that they apply to the whole of the *BankSystem* (**Result 5**). The results of these checks are shown in Figure 9. We see that this first version succeeds on only one check.

The checks mostly fail because all the important state is hidden within B but some state is required in the CSP to ensure that the properties hold for the controller processes (using FDR). This need for some state information in

the CSP controllers is clearly exhibited when we fail to show that $SPEC \sqsubseteq_T BankSystemControllers\,(version1)$ because the controllers allow more behaviours than the specification permits. Consider the following trace

$$\langle enterBank.c1, report.success, setCustomer.c1, getCustomer.c2, leaveBank.c2\rangle$$

which illustrates the customer $c2$ attempting to leave the bank without having first entered it. This trace is not allowed by the process $SPEC$ but it is permitted by $BankSystemControllers$. This is because the channel $getCustomer$ accepts any cc as input (since no guards are enforced) and so $leaveBank.c2$ is a visible event when it should not be.

A similar counter example can be identified when we attempt to verify $SPEC2 \sqsubseteq_F BankSystemControllers(version1)$. The root cause of the verification failure is again the complete hiding of state purely within B.

Note, in the table, we do not check for the stable failures refinement of $SPEC$ against the $BankSystemControllers$ in any of the development versions because it would require the bank system to allow users to use the bank completely independently. For example, after $enterBank.c1$, the event $enterBank.c2$ is still possible in $SPEC$ (due to the interleaving). However, the $BankSystemControllers$ definition refuses it, because it will not allow anything else until a report is provided. This is perfectly reasonable, the bank system is not expected to process customers completely independently. Hence, $SPEC$ as a liveness specification ($SPEC \sqsubseteq_F BankSystemControllers$) is unreasonable.

4.2 Version 2 of the Bank System

In the previous version we noted that retrieving any customer, cc, along the channel $getCustomer$ does not restrict the CSP behaviour appropriately, but there was no state in the CSP which could be used to express such a restriction on that communication allowed. Therefore, in this second version we re-design $BankSystemControllers$ to include some useful state. We begin by representing the $currentCustomer$ as state within the CSP description. One option is to duplicate the state in both the B and the CSP (as we had shown in our example in [10]). In this paper we minimise duplication, and so we do not capture $currentCustomer$ in the $Counter$ machine. Given this change to the $Counter$ machine we note that the only other variable in this machine is $queueNo$. This number is not used within $CounterCtrl$ to restrict the flow of control and begs the question, why capture it in that part of the system? Would it not be more appropriate in the queues partition of the system? We had initially considered moving it to the $Queues$ machine but since it is only a natural number it can be easily captured solely in the $QueuesCtrl$. This means that we no longer need the $com2$ channel to communicate the queue number across the distributed system. Furthermore, $com3$ is simplified to only passing the next customer to be serviced along its channel. Thus, the overall simpler architecture of the second version is shown in Figure 10. The $Counter$ machine is no longer needed but the $Queues$ machine contains complex state and remains almost unchanged (as we shall discuss below).

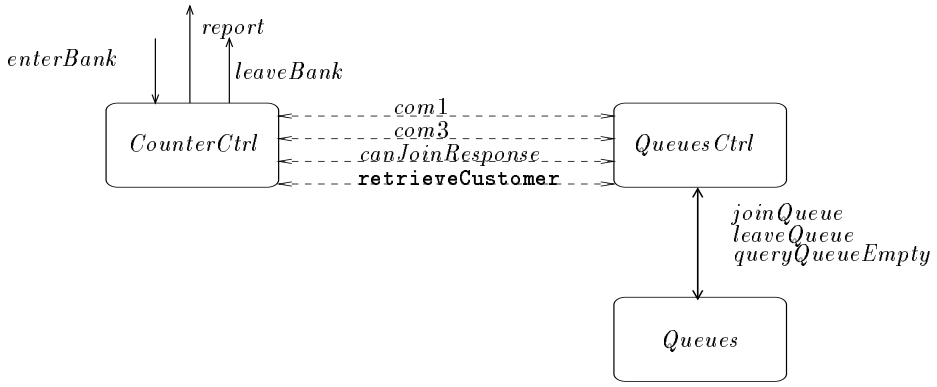

Fig. 10. Communication Architecture of *BankSystem*- Version 2

The second version of the *BankSystemControllers* is defined fully in Figure 11. The processes have similar execution paths but utilise fewer B query and setting operations. This version achieves all the verification aims of **Steps 3,4,6** for the system, but was achieved after some iteration. Our first attempt at **Step 4** showed the controllers reaching a deadlock situation. It was possible for the *CounterCtrl* to have processed two customers and would then be attempting to allow another person to join the queues by sending a communication along $com1$. The *QueuesCtrl* on the other hand had already resolved its choice internally and was attempting to allow a customer to leave but was forced to wait for a communication along $com2$. Thus, both synchronisation channels were waiting for each others' co-operation to continue execution. This was caused because we had limited the external interface of the bank system and so the CSP controllers were allowed to make internal progress when they should not be doing so. Therefore, we introduced a new synchronisation channel `retrieveCustomer` on one of the problematic branches to prevent this undesirable progress. Hence, $BankSystemControllers(version2)$ can now be shown to be deadlock-free. Our first attempt at **Step 5** highlighted the fact that we could not again prove $SPEC \sqsubseteq_T BankSystemControllers(version2)$. As with the first version this was due to the fact that we could allow customers to leave the bank who had not entered. This problem was solved by introducing *custSet* as extra state in the CSP, which is an abstraction of the customers held in queues. Then only customers who had entered the bank could proceed from queues to the counter, and subsequently leave. Hence we can show that the safety property is preserved. Carrying around this extra information in the CSP had an impact on the first branch of the *QueuesCtrl*. The precondition of *joinQueue* states that the joining customer *cc* cannot already be in any queue. In Version 1, we guaranteed this by first calling a query operation. In this version we can make use of *custSet* and use this as an alternative way of checking that *cc* is not already a member of this set. Consequently, we do not need to use the *queryIsInQueue*

$CounterCtrl = CurrentCtrl(0, defaultCustomer)$
$CurrentCtrl(num, currentCust) =$
$\quad JoinCtrl(num, currentCust) \;\Box\; LeaveCtrl(num, currentCust)$

$JoinCtrl(num, currentCust) =$
$\quad num < maxLimit \,\&\, enterBank?cc \rightarrow$
$\quad ((num = 0 \,\&\, report.success \rightarrow CurrentCtrl(num + 1, cc))$
$\quad\;\Box$
$\quad ((num > 0 \,\&\, \neg(cc = currentCust)) \,\&\, com1!cc \rightarrow$
$\qquad canJoinResponse?bb \rightarrow report.bb \rightarrow$
$\qquad$ **if** $(bb = success)$
$\qquad$ **then**
$\qquad\quad CurrentCtrl(num + 1, currentCust)$
$\qquad$ **else**
$\qquad\quad CurrentCtrl(num, currentCust))$
$\quad\Box$
$\quad ((num > 0 \& (cc = currentCust)) \,\&\, report.fail \rightarrow CurrentCtrl(num, currentCust)))$

$LeaveCtrl(num, currentCust) =$
$\quad (num = 1 \,\&\, leaveBank!currentCust \rightarrow CurrentCtrl(0, defaultCustomer))$
$\quad\Box$
$\quad (num > 1 \,\&\, leaveBank!currentCust \rightarrow$
$\qquad retrieveCustomer \rightarrow com3?cc- > CurrentCtrl(num - 1, cc))$

$QueuesCtrl = QCtrl(0, 1, \varnothing)$
$QCtrl(s, queueNo, custSet) =$
$\quad (s < maxQueueingCustomers \& com1?cc \rightarrow$
$\qquad$ **if** $(cc \in custSet)$
$\qquad$ **then**
$\qquad\quad canJoinResponse!fail \rightarrow QCtrl(s, queueNo, custSet)$
$\qquad$ **else**
$\qquad\quad canJoinResponse!success \rightarrow joinQueue.cc \rightarrow$
$\qquad\quad QCtrl(s + 1, queueNo, custSet \cup \{cc\}))$
$\quad\Box$
$\quad (s > 0 \,\&\, retrieveCustomer \rightarrow NextQCtrl(s, queueNo, custSet))$

$NextQCtrl(s, queueNo, custSet) =$
$\quad queryQueueEmpty!queueNo?bb \rightarrow$
$\quad$ **if** $(bb = no)$
$\quad$ **then**
$\qquad leaveQueue!queueNo?\{cc : custSet\} \rightarrow com3!cc \rightarrow$
$\qquad QCtrl(s - 1, inc(queueNo), custSet - \{cc\}))$
$\quad$ **else**
$\qquad NextQCtrl(s, inc(queueNo), custSet)$
$A = \{|\, com1, com3, canJoinResponse, retrieveCustomer \,|\}$
$B = \{|\, joinQueue, leaveQueue, queryQueueEmpty \,|\}$
$BankSystemControllers = CounterCtrl[|\, A \,|]\, QueuesCtrl \setminus \cup(A, B)$

Fig. 11. *BankSystemControllers* - Version 2

$NextQCtrl(s, queueNo, lastQueueNo, custSet) =$
$(queryQueueEmpty!queueNo?bb\langle queueNo = lastQueueNo \Rightarrow bb = no\rangle \rightarrow$
if $(bb = no)$
then
$leaveQueue!queueNo?\langle cc \in custSet\rangle \rightarrow com3!cc \rightarrow$
$QCtrl(s-1, inc(queueNo), custSet - \{cc\})$
else
$NextQCtrl(s, inc(queueNo), lastQueueNo, custSet))$

Fig. 12. Extra state in *NextQCtrl* to prevent livelock - Version 3

operation from the *Queues* machine in this version (which is why it is missing from Figure 10). Adding all the extra state meant that we had to re-do the initial step (1-6). Now, ensuring divergence-freedom of $QueuesCtrl \parallel Queues$ is more involved because the relationship between the CSP and B (captured in the *CLI*) is as follows;

$$s = \Sigma_{i \in dom(customerQueues)} card(customerQueues(i)) \wedge$$
$$custSet = ran(\bigcup(ran(customerQueues)))$$

4.3 Version 3 of the Bank System

Version 2 above does not ensure divergence-freedom of the *BankSystem* when examining the CSP in isolation (**Step 5**). It is not possible as it stands because the assumption ($\{cc \in custSet\}$) in the CSP description introduces a possible divergence. An application of **Result 4** enables us to transform the assumption into a guard to obtain an equivalent controller for the B machines. Having removed this assumption the only remaining way divergence can arise is from an internal loop.

In fact, as in Version 1, the CSP description does allow an internal loop in *NextQCtrl*, forever obtaining $queryQueueEmpty!queueNo.yes$ on every *queueNo* input in turn. However, *NextQCtrl* will always terminate because we only call it when $s > 0$, and so there is at least one non-empty queue which will give a result $queryQueueEmpty!queueNo.no$ and exit the internal loop. We need to include this information into the CSP controller if we wish to establish results purely from checking the CSP part of the system.

One way to achieve this is by introducing an additional item of state in order to express a guard on the values provided by the *Queues* machine: the last queue that will have to be checked in order to obtain a *no* result. When entering the loop, this value is set to be the queue preceding the first queue to be checked — this will be reached only if all other queues are empty. This change is given in the new *NextQCtrl* process, defined in Figure 12. The process is called within *QCtrl* with the parameters $(s, queueNo, dec(queueNo), custSet)$. The guard enforces that if the last queue is being queried the answer must be *no*.

This extra state and guard is enough to ensure that *NextQCtrl* does not loop infinitely, and indeed $BankSystemControllers(version3)$ is livelock-free. It is also necessary to show that this version of the controllers, with a guard in place of the assumption, is still consistent with the controlled machines. This is straightforward to achieve: it essentially includes within the *CLI* that some queue between *queueNo* and *lastQueueNo* is non-empty. Having established this, we can then drop the guard completely, obtaining a controller whose behaviour does not depend on the value of *lastQueueNo* at all, and which is equivalent to Version 2. **Result 6** then allows this part of the state to be dropped, arriving back at (the equivalent) Version 2 of *BankSystemControllers* again. Thus, the system controlled by Version 2 must also be divergence-free.

This completes the verification of the *BankSystem* and so to summarise - Version 1 was fine if we were only concerned with deadlock freedom but if we wanted to show that the system exhibited some more interesting properties and also divergence freedom we needed Version 2. Version 3 was only introduced to assist in the verification of Version 2.

5 Discussion

The paper identified general development steps so that our (CSP||B) approach can be adopted more widely. The immediate benefit of adoption is the availability of tool support for almost all of these steps. Ongoing research is being conducted to provide tool support for **Step 3** of the development process.

In this paper we significantly extended our approach so that a main external interface of a combined system can be identified, and a combined communicating system can be shown to exhibit some liveness and safety properties. We can also show deadlock and divergence freedom. The checking of these behavioural properties can be done using model-checking tools but the properties then hold for the combined communicating system. In the case study we demonstrated how state needed to be filtered up from the B into a CSP description so that we could prove the desired system properties. As a by-product, we showed that the architecture could be streamlined to contain only data-rich B components. We also showed that a B component is essential when complex state is involved because such state cannot be described easily within CSP.

To the best of our knowledge there are only two other approaches which aim to verify that a communicating specification meets some abstract specification. Circus [2] is one such approach which begins with an abstract specification and by applying strict refinement laws a combined specification (using Z and CSP) can be defined. Hence, they elegantly show by construction that a combined specification meets an abstract specification. Muntean and Rolland [7] take a similar approach but all the communicating interaction is expressed purely within B. Both approaches are tool supported but do not allow model checking of behavioural properties.

Acknowledgements Thanks to Lok Yeung for his careful reading of earlier drafts, and to Jim Woodcock and Anna Cavalcanti for a useful discussion which helped to clarify some issues regarding Circus.

References

1. Abrial J. R.: *The B Book: Assigning Programs to Meaning*, CUP (1996).
2. Cavalcanti A., Sampaio A., and Woodcock J.: *Refinement of Actions in Circus*, In REFINE'02, FME Workshop, Copenhagen (2002).
3. Hoare C. A. R.: *Communicating Sequential Processes*, Prentice Hall (1985).
4. Morgan C. C.: *Of wp and CSP.* In W.H.J. Feijen, A.J.M. van Gasteren, D. Gries and J. Misra, editors, *Beauty is our business: a birthday salute to Edsger W. Dijkstra.* Springer (1990).
5. Formal Systems (Europe) Ltd.: *Failures-Divergences Refinement: FDR2 User Manual* (1997), `http://www.formal.demon.co.uk`
6. Neilson D., Sorensen I. H.: *The B-Technologies: a system for computer aided programming*, B-Core (UK) Limited, Kings Piece, Harwell, Oxon, OX11 0PA (1999), `http://www.b-core.com`
7. Muntean T., Rolland O.:*Distributed Refinement: application to the B Method*, RCS'02, International Workshop on Refinement of Critical Systems: Methods, Tools and Experience, Grenoble (2002), `http://www.esil.univ-mrs.fr/~spc/rcs02/rcs02.html`.
8. Schneider S.: *Concurrent and Real-time Systems: The CSP approach*, Wiley (2000).
9. Schneider S.: *The B-Method: An Introduction*, Palgrave, 2001.
10. Schneider S.,Treharne H.: *Communicating B Machines.* ZB2002, Grenoble, LNCS 2272, Springer, January (2002).
11. Schneider S.,Treharne H.: *CSP Theorems for Communicating B Machines.* Technical Report CSD-TR-02-12, Department of Computer Science, Royal Holloway (2002).
12. Treharne H., Schneider S.: *Using a Process Algebra to control B OPERATIONS.* In K. Araki, A. Galloway and K. Taguchi, editors, IFM'99, York, Springer (1999).
13. Treharne H., Schneider S.: *How to drive a B Machine.* ZB2000, York, LNCS 1878, Springer, September (2000).
14. Treharne H.: *Controlling Software Specifications.* PhD Thesis, Royal Holloway, University of London (2000).
15. Treharne H., Schneider S.: *Communicating B Machines (full version).* Technical Report, RHUL (2001).