Communicating B Machines

Steve Schneider and Helen Treharne

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

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

Abstract. This paper describes a way of using the process algebra CSP to enable controlled interaction between concurrent B machines. This approach supports compositional verification: each of the controlled machines, and the combination of controller processes, can be analysed and verified separately in such a way as to guarantee correctness of the combined communicating system. Reasoning about controlled machines separately is possible due to the introduction of guards and assertions into description of the controller processes in order to capture assumptions about other controlled machines and provide guarantees to the rest of the system. The verification process can be completely supported by different tools. The use of separate controller processes facilitates the iterative development and analysis of complex control flows within the system. The approach is motivated and illustrated with a non-trivial running example.

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

1 Introduction

This paper introduces a new approach to combining concurrent B machines in a verifiable way. One of the main motivations for the approach is the desire to make use of existing tool support for all aspects of the verification, and for the generation of executable code within the B-Method. This builds on previous work [12–14] using the process algebra CSP [6] to describe controllers for B machines, in order to express and reason about complex flows of control in a natural way. Previous work has been concerned with a single *controller* process P encapsulating a single flow of control for a B *machine* M; M can be a single machine or be comprised of a hierarchy of machines. We also focussed on methods for proving that a controller is consistent with its underlying machine. In this paper we consider how a collection of such combinations (of B machines M_i and their controllers P_i) can interact. The controllers play a key role in enabling communication between the machines. We propose an architecture (pictured in Figure 1) which forces all interaction between machines to be through the controllers. The architecture is therefore appropriate both when controlled machines

are distributed across a network and when they are executed on the same processor. This enables deadlock and divergence freedom of a *combined communicating system* to be verified in a compositional way, simply by analysing smaller parts of the system: the individual *controlled machines* $P_i \parallel M_i$, and the parallel combination of only the P_i (without the M_i). Overall correctness follows automatically from Lemma 1 and Theorem 2 presented in Section 4.2. This means that existing tools for the B-Method and for process algebra can be used to verify our system descriptions. Other properties of the combined communicating system can also be specified within the process algebra and established using existing tools.

In practice, we find that the correctness of particular controlled machines within the system rests on the behaviour of other controlled machines, and aspects of the rest of the system's behaviour must be incorporated into the verification. We achieve this by extending the language of controllers to include *guards* (which block unwanted inputs) and *assertions* (which diverge on unexpected communications). Guards on an input channel of a controller P are used to describe the inputs expected from the rest of the system, and thus capture the assumptions about the process' environment. This enables P to be analysed in the absence of the rest of the system. Assertions on outputs to the rest of the system describe what the process itself should guarantee. Assertions are also used to encapsulate the expectations on inputs to a controller P from its associated machine M. This enables the combination of controllers to be analysed together, independently of the machines they control, and hence entirely at the level of process algebra.

The development of a combined communicating system proposed in this paper will involve the following steps:

1. Define the individual B machines,
2. Give CSP controllers for them that describe the flow of control for their use,
3. Prove consistency between the B machines and their controllers,
4. Prove deadlock freedom of the combination of the controllers,
5. Refine and implement the machines and controllers independently.

This process will be illustrated by the running example.

This paper is organised as follows: Section 2 introduces the CSP controller language and semantics; Section 3 describes the running example used to motivate and illustrate the approach; Section 4 is concerned with the consistency results which underpin the approach, and the use of assertions and guards; 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

Communicating Sequential Processes (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 synchronise. Events can be unstructured (such as *start*), or they can have some

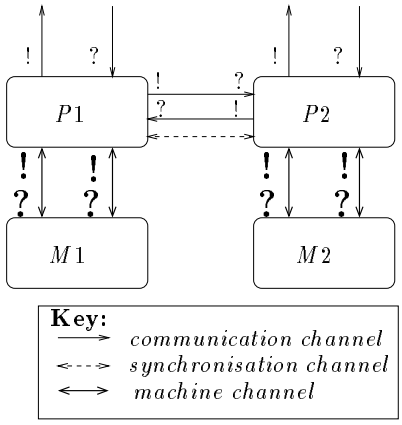

Fig. 1. An architecture for concurrent B machines

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 [13, 14] and is given by the following pseudo-BNF rule;

$$P ::= a \rightarrow P \mid c?x\langle E(x)\rangle \rightarrow P \mid d!v\{E(v)\} \rightarrow P \mid e\,\mathbf{?}\,v\,\mathbf{!}\,x\{E(x)\} \rightarrow P \mid$$

$$P_1 \,\Box\, P_2 \mid P_1 \sqcap P_2 \mid \prod_{x|E(x)} P \mid \mathit{if}\ b\ \mathit{then}\ P_1\ \mathit{else}\ P_2\ \mathit{end} \mid S(p)$$

where $a \in \Sigma$ and is a *synchronisation event*, c is a *communication channel* accepting inputs, d is a communication channel sending output values, 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.

All the terms in the above grammar have established CSP semantics and can be modelled in a CSP model checker such as FDR [8] without any extensions. We will explain each expression shortly but before doing so we need to clarify our use of channels.

In our language we want to distinguish between communication via individual CSP processes and the inter-communication of CSP processes with their underlying B machines, as shown in Figure 1.

Synchronisation events and communication channels serve as communication mediums between two CSP controllers and they have *no* correspondence with B operations. Conversely, a machine channel corresponds to a B operation, with

the same types for inputs and outputs. The machine channel comprising of input and output also admits degenerate cases, in which either or both the $?$ and $!$ may be dropped. These cases are all treated as special instances of the machine channel, and do not require separate treatment. For example, a machine channel with no input and outputs will simply correspond to a B operation with no parameters.

The expression, $a \rightarrow P$, means that a process is prepared to engage in the event a, and subsequently behave as P. It is used to provide a way of synchronising on atomic events.

Another expression which makes use of the prefix operator ($\rightarrow$) is the expression $c?x\langle E(x)\rangle \rightarrow P$. It denotes a process that is initially prepared to accept any value x on the communication channel c that meets the guard predicate $E(x)$, after which it behaves as the process P (which may depend on the value of x) but it will not accept any x that fails to meet that predicate. Guards can be modelled in FDR using standard CSP syntax.

The expression $d!v\{E(v)\} \rightarrow P$ is initially prepared to perform the event $d.v$ (i.e. pass values v along the communication channel d). If the assertion $E(v)$ is true, then its subsequent behaviour is that of P; if $E(v)$ is false then it *diverges*. Divergent behaviour will be discussed below. Assertions are simply coded in FDR as follows;

$$d!v\{E(v)\} \rightarrow P = d!v \rightarrow (\mathit{if}\ E(v)\ \mathit{then}\ P\ \mathit{else}\ DIV)$$

where DIV represents a divergent process.

The expression $e?v!x\{E(x)\} \rightarrow P$ is initially prepared to allow a process to interact on machine channel e. This channel will be used to communicate with a B machine via its corresponding operation $x \longleftarrow e(v)$, so it provides v as input to the B machine (indicated by the $?$), and accepts x as output (indicated by the $!$). If the value x it receives meets the predicate $E(x)$ then it behaves as the process P which may depend on the value of x, otherwise it diverges. Observe that the CSP semantics of this term will be the same as an event communicating over an output and input channel in standard CSP (for example $e!v?x\{E(x)\}$).

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 ($\bigsqcap$) 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 a recursive family indexed by sets:

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

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.

Note that the difference with the above language and the one used in our previous work is the inclusion of an output communication channel (!), internal choice ($\sqcap$), and indexed internal choice ($\bigsqcap$). Furthermore, the use of the input channel ? and the atomic event a have been restricted to describing communication between CSP controllers, and they no longer correspond to underlying B operations. We have also augmented machine and communication channels with assertions and guards. This rationalisation of the notation is necessary because we need to deal with concurrency cleanly. In this paper we have also restricted the language to non-terminating controllers; but the inclusion of terminating loops is discussed in [15].

In addition to the language for controllers, CSP provides a number of other operators, including *parallel composition*: $P_1 \parallel P_2$. There is also an indexed form $\|_i\ P_i$. A parallel combination 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.

For example, if the processes *Copy* and *Copy*2 are defined as follows

$$Copy = in?x \rightarrow out!x \rightarrow Copy$$
$$Copy2 = out?y \rightarrow down!y \rightarrow Copy2$$

then $Copy \parallel Copy2$ can input values v on *in*, have both components synchronise on $out.v$ which passes the value to *Copy*2, and then have *Copy*2 independently output v on *down*.

2.2 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 [10].

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 CSP model-checker FDR allows checks for deadlock and divergence freedom to be made automatically for processes.

3 Example

We describe and motivate the approach of this paper through an illustrative example. This section illustrates the use of the above control language in specifying CSP controllers to drive underlying B machines. Each individual controller acts as an interface for its underlying machines and provides a possible pattern of execution. The aim is to compose a collection of controlled machines in order to specify a large combined communicating system.

For the purposes of presentation we first describe the machines and then the associated controllers. In practice, we develop both alongside each other, since both impact on each other's specification as we shall see in Section 3.2 when we discuss the particular controllers. The B-Toolkit [9] and FDR source files for the example can be downloaded at
`http://www.cs.rhul.ac.uk/home/helen/papers/zb2002/sources.tar.gz`

3.1 Machines

Consider two B machines, *CustomerData* and *CheckoutData*, which are defined in Figures 2 and 3 respectively. The purpose of these machines is to capture information about customers who are shopping and process them through checkout queues in a supermarket. The separation of customers and checkout processing illustrates our approach of modelling different parts of the specification separately.

The customer information is captured in the *CustomerData* machine. It introduces the variable *customer* to track whether an individual customer is either *shopping* or *paying* (i.e. the customers' status). The set of all possible customers, *CUSTOMERS* is declared in a separate context machine, *Types*, as shown in Figure 4. Similarly, the set defining the customers' status, *STATUS*, is declared in the same context machine. This allows more than one machine to use static information and is typical of B developments.

Four operations are offered by *CustomerData*: *beginshopping* allows a customer to become a shopper; *whatstate* queries the status of a customer; *proceedtocheckout* updates a customer from being one who is shopping to one who is

```
MACHINE CustomerData
SEES Types
VARIABLES customer
INVARIANT customer ∈ CUSTOMERS ⇸ STATUS
INITIALISATION customer := ∅

OPERATIONS
   beginshopping ( cust ) ≙
    PRE cust ∈ CUSTOMERS ∧ customer ( cust ) ≠ paying THEN
      customer ( cust ) := shopping
    END ;
  ss ⟵ whatstate ( cust ) ≙
    PRE cust ∈ CUSTOMERS THEN
      IF customer = ∅ THEN ss := idle ELSE
        ss := customer ( cust )
      END
    END ;
   proceedtocheckout ( cust ) ≙
    PRE cust ∈ CUSTOMERS ∧ customer ( cust ) = shopping THEN
      customer ( cust ) := paying
    END ;
   finished ( cust ) ≙
    PRE cust ∈ CUSTOMERS ∧ customer ( cust ) = paying THEN
      customer := { cust } ⩤ customer
    END
END
```

Fig. 2. *CustomerData* Machine

waiting to pay; and *finished* removes a customer from being considered by the system (once goods have been paid for).

The queue of customers is tracked in the *CheckoutData* machine[1]. This machine has a number of checkout counters, some of which will be open with a queue of customers. It introduces two variables, *opencounters* and *queues*. The variable *opencounters* keeps track of the counters that are currently open. The set of all possible counters, *COUNTERS*, is again defined in the separate context machine, *Types*. Only counters that are open can have a queue of customers associated with them and this information is captured by the *queues* variable. The invariant of the machine provides constraints on a queue of customers, stating that customers should only ever appear in at most one queue, and in at most one position.

Six operations are offered by the machine. The operation *do_open* opens a new counter provided that not all counters are already open. The operation *do_close* closes a counter and removes any customers who are queueing at that counter. The operation *serve* processes the customer at the head of the queue at a given counter. The operation *join* allows a new customer to join the queue at a particular open counter. The operation *drop_out* allows a customer to leave a queue. The operation *choose* non-deterministically selects an open counter whenever possible.

[1] originally inspired by an example in [11]

```
MACHINE CheckoutData
SEES Types , Bool_TYPE
VARIABLES queues , opencounters
INVARIANT
opencounters ⊆ COUNTERS ∧
queues ∈ opencounters → iseq ( CUSTOMERS ) ∧
∀ ( cc , dd ) . ( cc ∈ COUNTERS ∧
dd ∈ COUNTERS ∧ cc ≠ dd ⇒
ran ( queues ( cc ) ) ∩ ran ( queues ( dd ) ) = ∅ )
INITIALISATION
queues := ∅ || opencounters := ∅

OPERATIONS
rep_co ←– do_open ≙
  BEGIN
  IF opencounters ≠ COUNTERS THEN
    ANY co
    WHERE co ∉ opencounters
      ∧ co ∈ COUNTERS THEN
      opencounters := opencounters ∪ { co } ||
      queues ( co ) := [ ] || rep_co := co
    END
  ELSE
    rep_co := def_counter
  END
  END ;
rep_set ←– do_close ≙
  BEGIN
  IF opencounters ≠ ∅ THEN
    ANY co
    WHERE co ∈ opencounters THEN
      rep_set := ran ( queues ( co ) ) ||
      queues := { co } ⩤ queues ||
      opencounters := opencounters − { co }
    END
  END
  END ;
cu , rep ←– serve ( co ) ≙
  PRE co ∈ COUNTERS THEN
  IF co ∈ dom ( queues ) ∧
    queues ( co ) ≠ [ ] THEN
    cu := first ( queues ( co ) ) ||
    queues ( co ) := tail ( queues ( co ) ) ||
    rep := TRUE
  ELSE
    rep := FALSE ||
    cu :∈ CUSTOMERS
  END
  END ;
join ( co , cu ) ≙
  PRE cu ∈ CUSTOMERS ∧
    co ∈ COUNTERS ∧
    co ∈ opencounters∧
    cu ∉ ⋃ cc .
    ( cc ∈ dom ( queues ) |
        ran ( queues ( cc ) ) )
  THEN
  queues ( co ) := queues ( co ) ← cu
  END ;
drop_out ( cu ) ≙
  PRE cu ∈ CUSTOMERS THEN
    ANY co
    WHERE co ∈ COUNTERS ∧
      cu ∈ ran ( queues ( co ) ) THEN
      LET ss
      BE ss = queues ( co ) IN
        queues ( co ) :=
        ss ↑ ss ⁻¹ ( cu ) − 1 ⌢
        ss ↓ ss ⁻¹ ( cu )
      END
    END
  END ;
out_co , rep ←– choose ≙
  BEGIN
    IF opencounters = ∅ THEN
      rep := FALSE ||
      out_co := def_counter
    ELSE
      ANY cc
      WHERE cc ∈ opencounters
      THEN
        rep := TRUE ||
        out_co := cc
      END
    END
  END
  END
```

Fig. 3. *Checkout* Machine

MACHINE *Types*
SETS $COUNTERS$; $CUSTOMERS$; $STATUS = \{ shopping, paying, idle \}$
CONSTANTS $def\_counter$
PROPERTIES $COUNTERS = 1 .. 10 \wedge CUSTOMERS = 1 .. 100 \wedge$
$def\_counter \in COUNTERS$
END

Fig. 4. *Types* Machine

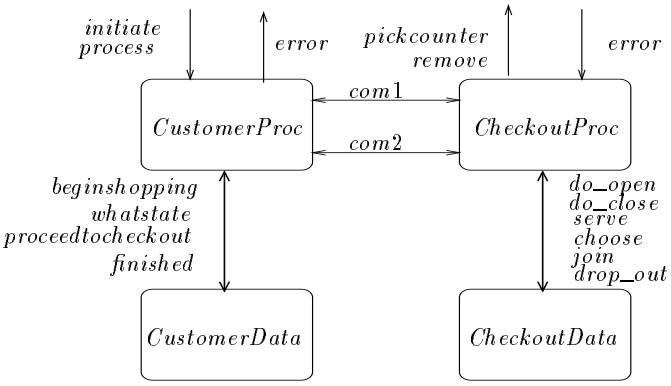

Fig. 5. Example Architecture

3.2 Controllers

In order to compose two B machines into a combined communicating system we introduce CSP controllers for each of them which communicate with one another. To be consistent with the B machines the controllers must ensure that operations are always called within their preconditions. In this section we present the controllers which drive the *CustomerData* and *CheckoutData* machines. Specifying the controllers, *CustomerProc* and *CheckoutProc* below, allows us to build the system $((CustomerProc \parallel CustomerData) \parallel (CheckoutProc \parallel CheckoutData))$. Figure 5 illustrates the overall architecture of the whole system and highlights all the channels involved.

CustomerProc is a controller which deals with customer requests, and is given in Figure 6. The process *CustomerProc* is defined in terms of a parameterised recursion. The parameter *custset* holds the set of customers currently paying. Initially, there will be no such customers and the set is empty. Note that the state carried around in this controller is more abstract than the state in the underlying B machine. In practice, our aim is to minimise the amount of state in the CSP controller. However, some state may be necessary in order to determine the flow of control in a process and/or for use in specifying assertions and guards, as is the case with *custset*.

$$
\begin{aligned}
CustomerProc &= Q1(\varnothing) \\
Q1(custset) &= (initiate?cc\langle cc \in CUSTOMERS - custset\rangle \rightarrow beginshopping\,?\,cc \rightarrow \\
&\quad Q1(custset)) \\
&\quad \Box\ (process\,?cc \rightarrow whatstate\,?\,cc!\,ss \rightarrow \\
&\qquad\qquad if\,(ss == shopping)\,then \\
&\qquad\qquad\quad (proceedtocheckout\,?\,cc \rightarrow com1!cc\{cc \notin custset\} \rightarrow \\
&\qquad\qquad\quad Q1(\{cc\} \cup custset)) \\
&\qquad\qquad else \\
&\qquad\qquad\quad (error!cc \rightarrow Q1(custset))) \\
&\quad \Box\ (com2?cc\langle cc \in custset\rangle \rightarrow finished\,?\,cc \rightarrow Q1(custset - \{cc\}))
\end{aligned}
$$

Fig. 6. Customer Process

There are three main paths of execution which are controlled by external choice. The first path accepts customers provided they are not already paying. This is ensured by the guard of the communication channel *initiate*, ($cc \in CUSTOMERS - custset$). We proceed after accepting an appropriate customer to observe a communication of that customer value along the *beginshopping* channel which corresponds to the underlying B operation assigning his/her status to be *shopping*. Allowing only certain customers means that the precondition of the *beginshopping* operation will be discharged when invoked. It is true that the above constraint on customers could have been modelled as a *select* statement in B but our philosophy is never to block B operations, so that we can implement the operations in ANSI C within the B-Toolkit. The role of guards and assertions will be examined in more detail in Section 4.3.

The second path processes a customer. If the customer is already shopping then the underlying B state of that customer is updated followed by a message being passed along $com1$ communicating to *CheckoutProc* that the customer can now join a queue. Note that the assertion denotes the assumption that the customer is not already paying. Note also that the parameter in the recursive call updates the *custset* to include the new customer. In the case where the customer is not shopping an error is reported and no underlying B state is affected.

Observe that a request to process a customer could occur even before he/she has begun shopping and thus impacting on the specification of the *whatstate* operation. Initially, we had used a simple assignment and a precondition stating that $cust \in dom(customer)$. The operation in Figure 2 is far more robust and allows the operation to be called with any customer as its input. In order to achieve this particular style of specification we augment the datatype $STATUS$ to include an extra enumeration, $idle$, as shown in Figure 4.

The third path records the fact that a customer has finished shopping. The particular customer, whose status is to be updated, will be communicated along $com2$. The guard on the channel reflects our understanding that the customer will be one who is currently paying. Here again the guard is used to dischargee

$$
\begin{array}{l}
CheckoutProc = Q2(\varnothing) \\
Q2(custque) \ = (do\_open!out\_co \rightarrow Q2(custque)) \\
\qquad \Box\ (do\_close!out\_que\{out\_que \subseteq custque\} \rightarrow Q2(custque - out\_que)) \\
\qquad \Box\ (\bigsqcap_{d \in COUNTERS} pickcounter!d \rightarrow \\
\qquad\quad serve?d!cust!rep\{(cust \in custque) \vee (rep == false)\} \rightarrow \\
\qquad\quad if\ (rep == true)\ then \\
\qquad\qquad (com2!cust \rightarrow Q2(custque - \{cust\})) \\
\qquad\quad else \\
\qquad\qquad Q2(custque)) \\
\qquad \Box\ (com1?cust\langle cust \in (CUSTOMER - custque)\rangle \rightarrow \\
\qquad\quad choose!out\_co!rep \rightarrow \\
\qquad\quad if\ (rep == true)\ then \\
\qquad\qquad join?out\_co?cust \rightarrow Q2(custque \cup \{cust\}) \\
\qquad\quad else \\
\qquad\qquad do\_open!out\_co \rightarrow join?out\_co?cust \rightarrow Q2(custque \cup \{cust\})) \\
\qquad \Box\ (remove?cc \rightarrow \\
\qquad\quad if\ (cc \in custque)\ then \\
\qquad\qquad drop\_out?cc \rightarrow com2!cc \rightarrow Q2(custque - \{cc\}) \\
\qquad\quad else \\
\qquad\qquad error!cc \rightarrow Q2(custque))
\end{array}
$$

Fig. 7. Checkout Process

the associated precondition in the *finished* operation. The set substraction in the recursive call keeps the set of paying customers up to date.

CheckoutProc is a controller which controls the opening and closing of supermarket counters, and the processing of customers in queues associated with those counters. The process *CheckoutProc* is defined, in Figure 7, in terms of a parameterised recursion. The parameter *custque* holds the set of customers currently in a queue. Initially, there are no such customers.

There are five main possible execution paths in *CheckoutProc*. The first path effectively invokes the underlying B operation to open a counter if at all possible.

The second path closes a counter. The assertion on the output set *out_que* serves to affirm our belief that the customers in the set are queueing customers.

The third path illustrates how the choice of which counter will be serviced next is resolved by the controller. The information regarding counters is not captured in the controller (at this stage, though it may be included in a refinement at a later stage) and so it could be the case that an empty counter is passed as an input to the B operation *serve*. Therefore, the B operation needs to be robust enough to allow for this possibility. It achieves this by outputting a report value, indicating success or failure of the operation. The report value is used to control the flow of execution so that either *com*2 communicates that the customer was served successfully or no such communication occurs. Note that there is only one

assertion on the output values of the machine channel *serve*. This is essential since the *cust* value may not always be in the queueing set.

The fourth path allows a new customer to join a queue. A synchronisation along the communication channel $com1$ provides the particular customer wishing to join a queue. The guard on the input channel means that we never want to follow this path of execution if the customer is already in a queue. Once a customer is received we then proceed to choose a counter whose queue the customer can join. If there are no open counters then we must open one before the customer is allowed to join its queue. Otherwise he/she is simply appended to the queue of the chosen counter. The operation $join(co, cu)$ corresponding to $join?out\_co?cust$ has a precondition stating that the customer can only join provided he/she is not already present in any of the queues of the open counters. The guard on the communication channel $com1$ ensures that the *cust* value meets that precondition.

We originally specified the *do_open* operation with no parameters. During the development of the fourth main execution path we realised that an output parameter was necessary. Consider the following scenario, after performing a *choose* operation the control process reads the *rep* variable in order to decide whether an open counter has been successfully chosen or whether no counters are open and so a failure is reported by the operation. In the latter case we then proceed to non-deterministically choose to open a counter using the *do_open* operation and then allow the customer to join a queue. The problem is that the counter being passed to the *join* operation may not be the same as the one opened by the *do_open* operation. This inconsistency problem is solved by augmenting the *do_open* operation with an output parameter and passing this output value to the *join* operation. The repercussions of this change on process $Q2$ is that the *do_open* operation in the first branch of the choice needs to be consistent in its signature. In the first path the value being passed along the output channel is never used but this is allowable.

The fifth path of process $Q2$ either removes a customer from the queue or reports an error. If the customer is successfully removed from the queue then $Q2$ forces a synchronisation to occur on channel $com2$ so that the information about the particular customer is also updated, as we described above.

The above example demonstrates that specifying how the operations are used in controllers forces us to consider their interface and robustness very carefully. If the operations are not robust, that is they contain preconditions, then those preconditions must be discharged by the CSP controller. Furthermore, the controllers themselves enable us to clearly visualise the flow of information between controlled machines in a combined communicating system. In the example all interactions between *CustomerProc* and *CheckoutProc* have been along machine channels $com1$ and $com2$ passing values. However, in general the language, presented in Section 2.1, also allows pure synchronisations (such as a *start* event).

4 Consistency of Combined Communicating Systems

In this section we discuss how consistency, i.e. divergence and deadlock freedom, of combined communicating systems can be established in a compositional way. There are two separate steps involved in the verification process (steps 3 and 4 in the overall development process outlined in Section 1). First, in Section 4.1, we check that each individual controlled machine is divergence-free. Second, we check deadlock freedom of the combination of the individual controllers *without* their underlying machines. In the case of our example, these steps involve checking that $(CustomerProc \parallel CustomerData)$ and $(CheckoutProc \parallel CheckoutData)$ are both divergence-free, and then checking that $(CustomerProc \parallel CheckoutProc)$ is deadlock-free. These two steps can be supported by the B-Toolkit and FDR respectively. The results, presented in Section 4.2, verify that these two separate steps are enough to ensure deadlock and divergence freedom of a combined communicating system, in our case $((CustomerProc \parallel CustomerData) \parallel (CheckoutProc \parallel CheckoutData))$ divergence- and deadlock-free.

4.1 Consistency of Single Controlled Machine

When we consider a controller P for a machine M, we have previous results which establish when the CSP controller is appropriate for the B machine: it must not call operations which fail to meet their preconditions. This means that within the failures/divergences semantics for P and M, the parallel combination $(P \parallel M)$ must be divergence-free. The following is the essence of the result from [12, 14] giving a sufficient condition to guarantee divergence freedom:

Theorem 1. *If CLI (control loop invariant) is a predicate such that*

$$CLI \wedge I \Rightarrow [\{BBODY_{S(p)}\}]CLI$$

for each $BBODY_{S(p)}$ *in* P *then* $(P \parallel M)$ *is divergence-free.*

The theorem states that the invariant CLI need not necessarily hold after each individual operation but must hold at each recursive call of a controller. As in [12, 14], we define a maximum sequence of operations, $\{BBODY_{S(p)}\}$, to be the translation of a process expression associated with $S(p)$ up to and including reaching a recursive call. If such a sequence of operations can establish the CLI then we know that all the operations are called within their preconditions and terminate. Thus if the CLI holds for all such process expressions within P then we have a way of checking whether any machine M acting under its controller P can diverge.

The above result also holds for this paper's CSP controller language, which allows P to have additional channels which are independent of M; and to have assertions and guards on the values passed along channels. In order to ensure this we have to provide extra cases in the translation of process expressions to their corresponding sequences of operations and simplify some existing cases. For

example, the translation of communication channels which have no underlying operations from M are outlined as follows;

$$\{c?x\langle E(x)\rangle \rightarrow P\} = \mathbf{ANY}\ x\ \mathbf{WHERE}\ E(x)\ \mathbf{THEN}\ \{P\}\ \mathbf{END}$$
$$\{d!v\{E(v)\} \rightarrow P\} = \mathbf{PRE}\ E(v)\ \mathbf{THEN}\ \{P\}\ \mathbf{END}$$

Indexed internal choice $\sqcap_{x|E(x)} P$ is also translated using an **ANY** clause, and is akin to a guard on a communication channel. The fully formal treatment can be found in [15].

Note that the translation rules above reveal why the annotations of CSP channels were termed *assertions* and *guards*. We wanted to model predicates on inputs as guards which when violated would be identified in the CSP analysis in the second step of verification when checking deadlock freedom. On the other hand we wanted to model predicates on outputs as assertions which would be discharged (in step one) when checking the individual controlled machines, $P_i \parallel M_i$, using the above result.

In our example, an appropriate *CLI* for $(CustomerProc \parallel CustomerData)$ is $custset = customer^{-1}[\{paying\}]$. It states that the set carried in the parameter of the process will match the set of customers who are paying, and this will be true at the start of the loop and hold at each recursive call. A suitable *CLI* for $(CheckoutProc \parallel CheckoutData)$ is $custque = \bigcup cc.(cc \in dom(queues) \mid ran(queues(cc))$. It states that the set carried in the parameter will match the set of all customers in all of the queues. Again this will be true at the start of the loop and hold at each recursive call.

4.2 Consistency of Multiple Controlled Machines

Here we consider two B machines M_1 and M_2 under the control of their respective CSP controllers P_1 and P_2, whose overall architecture has been previously depicted in Figure 1. Formally, we have the following constraints on the alphabets of the controllers and machines:

- $\alpha(M_1) \subseteq \alpha(P_1)$
- $\alpha(M_2) \subseteq \alpha(P_2)$
- $\alpha(M_1) \cap \alpha(P_2) = \varnothing$
- $\alpha(M_2) \cap \alpha(P_1) = \varnothing$

Thus each M_i communicates only with its associated P_i, and the various P_i communicate with each other, and with the environment of the combination, on separate channels not involving the M_i.

The results below verify that in order to check deadlock and divergence freedom of the overall combination, only two steps are needed, i.e. it is sufficient to firstly check divergence-freedom of each controlled machine separately, and then to check deadlock freedom only of the combination of the CSP controllers.

Lemma 1. *If $P_1 \parallel M_1$ is divergence-free, and $P_2 \parallel M_2$ is divergence-free, then $(P_1 \parallel M_1) \parallel (P_2 \parallel M_2)$ is divergence-free.*

Proof. This follows immediately from the semantics of parallel composition in the failures/divergences model: parallel composition preserves divergence-freedom.

As noted above, previous work provides techniques for establishing divergence freedom of a $P_i \parallel M_i$.

Theorem 2. *If $(P_1 \parallel M_1)$ is divergence-free, and $(P_2 \parallel M_2)$ is divergence-free, and $(P_1 \parallel P_2)$ is deadlock-free in the stable failures model, then $(P_1 \parallel M_1) \parallel (P_2 \parallel M_2)$ is deadlock-free in the stable failures and failures/divergences model.*

Theorem 2 extends to any number of B machines to be combined in parallel. Thus, Theorem 2 is proved for the general case in the appendix A. The overriding principle is that each B machine should have its own CSP controller, and any interaction it has with its controller is private between them, and no other process participates in it. Communication between controlled machines takes place between the CSP controllers using channels which are independent of the B machines. Using this scheme, all that needs to be checked to establish overall divergence freedom is that each $P_i \parallel M_i$ is divergence-free; and all that then needs to be checked to establish overall deadlock freedom is that the parallel combination $\|_i P_i$ is deadlock-free. The above results also extend to divergence-free controllers which do not have underlying B machines, as we shall see in Section 4.4.

4.3 Role of Assertions and Guards

Assertions and guards allow the overall system to be split into pieces that can each be independently verified. They are used to carry the assumptions and guarantees to decompose the proof obligations on the overall system to proof obligations on small combinations of controlled machines within the system. Moreover, they make explicit the designer's understanding of why the controlled machines should behave correctly when combined.

The assertions on outputs (along machine and communication channels) give conditions that any output value must have. Since failure to meet such an assertion results in a divergence, a successful divergence freedom check of a $P_i \parallel M_i$ ensures that any output from P_i meets the assertion. Making such assertions too strong however will result in $P_i \parallel M_i$ failing to be divergence-free. For example, if we had included, in Figure 7, the simpler assertion $cust \in custque$ on the *serve* communication channel then $(CheckoutProc \parallel CheckoutData)$ would not be divergence-free; since we could not always guarantee that the output provided by the *serve* operation in *CheckoutData* met the assertion required by *CheckoutProc*.

Using assertions on machine and communication channels means that values failing such assertions will be ignored when $\|_i P_i$ is analysed for deadlocks in

the stable failures model, since a divergence never reaches a stable state in order to contribute to a deadlock.

Guards on inputs on communication channels are also used when checking a particular $P_i \parallel M_i$ for possible divergences. Without the guards on inputs to the P_is, some inputs which are not possible within the system as a whole might give rise to a divergence within some particular $P_i \parallel M_i$ when considered separately. Observe that without the guards and assertions in the example, the machines ($CustomerProc \parallel CustomerData$) and ($CheckoutProc \parallel CheckoutData$) would not have been divergence-free. For example, if we had omitted the guard on the *cust* input value along $com1$ in the fourth branch of the choice in $Q2$ (Figure 7) then the precondition of the *join* operation can be violated on some inputs.

Guards must also be taken into account in the deadlock freedom checks of the combined controllers. Making a guard too strong may result in a deadlock when checking $\|_i P_i$.

If the output assertion on a communication channel between two controllers is the same as (or stronger than) the input guard, then both assertion and guard may be dropped from the controllers once the verification process is complete. In other words, having proven that all messages passed around the system meet the appropriate conditions, they do not require checking at run-time and so they can be dropped from the controller implementations. In the case of *CustomerProc* this also means that we no longer need to keep track of the set of paying customers in the recursion and as a result we obtain a simpler controller.

Note that we have not included guards on input values along machine channels in our language. M is not analysed independently of P, so guards capturing properties of inputs from P do not need to be given explicitly.

4.4 Developing Controllers

In Section 3.2 we developed two controllers. In fact an analysis using FDR shows that ($CustomerProc \parallel CheckoutProc$) is not deadlock-free. In retrospect this is not surprising given the complex interactions that occurs between the controllers, and some further development of the controllers was necessary.

This parallel combination of processes can deadlock because *CustomerProc* can follow a path starting with a *process* event which leads to a point where it must synchronise with *CheckoutProc* on $com1$ to proceed. However, *CheckoutProc* can follow a path starting with *remove* or *serve* which lead to a point where it must synchronise with *CustomerProc* on $com2$ to proceed. If *CustomerProc* and *CheckoutProc* both follow these paths at the same time, then they can reach a state in which they cannot synchronise on either $com1$ or $com2$.

Therefore, if we are to guarantee deadlock freedom of the controllers, one solution is to introduce a *regulator* process REG in parallel, which ensures that only one of these paths is entered at any time. Once one process has entered such a critical path, the other process will be blocked from entering a critical path of its own until the relevant *com* has occurred. An appropriate REG for our example is given in Figure 8 and we can establish that ($CustomerProc \parallel CheckoutProc \parallel REG$) is deadlock-free (now step two of the verification process is

$$
\begin{aligned}
REG = {} & process?x \rightarrow com1?y \rightarrow REG \\
& \Box\ remove?x \rightarrow com2?y \rightarrow REG \\
& \Box\ pickcounter?x \rightarrow com2?y \rightarrow REG
\end{aligned}
$$

Fig. 8. Regulator Process (REG)

finished). The REG process has no assertions, no guards and hence is divergence-free. It also has no underlying B machines, or equivalently (to apply Theorem 2) it can be considered as having a vacuous B machine M_0 with no operations. Thus $REG = REG \parallel M_0$, which is therefore divergence-free. Therefore, all the controlled machines in our example pass step one of the verification process. In turn the overall example is divergence- and deadlock-free.

In general, if the REG process includes assertions and guards, they would be treated in exactly the same way as for controllers with underlying B machines. We would need to prove that REG was divergence-free using our previous result; any assertions and guards would have to hold simply by virtue of the CSP description.

Observe that the regulator process forces a synchronisation on the *pickcounter* communication channel. In the original version of *CheckoutProc*, in Figure 7, we started the branch which serves a customer with a communication along the *serve* machine channel. The introduction of the regulator process meant that we had to force the synchronisation on a communication channel and not on a machine channel, and so we introduced the *pickcounter* channel. This is important since we do not allow two controllers to control the same underlying B operation. Our approach is to couple a single controller and a machine together.

5 Discussion

The B-Method provides supports for the specification and development of software system requirements in terms of a collection of operations which the system must implement. It is ultimately necessary to have some way of describing the flow of control which directs operation calls. We have outlined one way in this paper. Alternative approaches include the direct introduction of actions (operations with blocking guards) within the B description [2]; or the use of a process algebra to describe a flow of control explicitly as a prelude to translation into B [5]. Both these approaches support the specification of such systems using tools. However, the main difference between their approaches and ours is the lack of a complete development path to executable code using existing tool support (due to *select* not being implementable). Our B machines can use the B-Toolkit to generate executable code. Preliminary investigations also show that CSP controllers can be naturally expressed in Java [15].

In this paper we have shown how the complex interactions between B machines and flow of control of their operations can be naturally expressed using

CSP. Through the observations made when presenting the paper's example it is clear that our approach to describing combined communicating systems is iterative and compositional. The benefit of compositionality is that the verification of controllers and their machines can be done separately. This compositionality is achieved by the addition of assertions and guards to the CSP controller language, and the distinct architecture used to build combined communicating systems.

A further benefit of our approach is the ability to refine each controller and machine separately whilst maintaining divergence and deadlock freedom. For example, we could refine the *CheckoutProc* controller to keep track of the open counters explicitly, and thus refining the internal choice to be a choice over the set of open counters.

In this paper we have focused on verifying deadlock freedom of controllers. Using tool support in the verification process was useful and highlighted the fact that it is not straightforward to specify deadlock-free controllers. Other requirements on the communication patterns of combined communicating system can also be checked using FDR. These are generally specified either in terms of the process algebra itself, or as a predicate on the traces of the system. This is the subject of current research.

Acknowledgements Thanks to Anna Fukshansky and Craig Saunders for comments on an earlier draft of this paper.

References

1. Abrial J. R.: *The B Book: Assigning Programs to Meaning*, CUP (1996).
2. Abrial J. R.: *Extending B without Changing it (for Developing Distributed Systems)*. In H. Habrias, editor, Proc. of the 1st B Conference, Nantes, France (1996).
3. Butler M. J.: *A CSP Approach to Action Systems*, D.Phil Thesis, Programming Research Group, Oxford University (1992).
4. Butler M. J.: *An Approach to the Design of Distributed Systems with B AMN*. In J. Bowen, M. Hinchey D. Till, editors, ZUM'97, Springer (1998), pp 223-241.
5. Butler M. J.: *csp2B: A Practical Approach to Combining CSP and B*, In J.M.Wing, J. Woodcock, J. Davies, editors, FM'99 World Congress, Springer (1999).
6. Hoare C. A. R.: *Communicating Sequential Processes*, Prentice Hall (1985).
7. 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).
8. Formal Systems (Europe) Ltd.: *Failures-Divergences Refinement: FDR2 User Manual* (1997), `http://www.formal.demon.co.uk`
9. 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`
10. Schneider S.: *Concurrent and Real-time Systems: The CSP approach*, Wiley (2000).
11. Schneider S.: *The B-Method: An Introduction*, Palgrave (to appear).
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).

A Proof of Theorem 2

Theorem 2 (general case). If $(P_i \parallel M_i)$ is divergence-free for each i, and $\|_i P_i$ is deadlock-free in the stable failures model, then $\|_i (P_i \parallel M_i)$ is deadlock-free in the stable failures and failures/divergences model.

The proof of this theorem requires some preliminary results about the failures of B machines M, and of controllers P.

We first establish that for any operation e of M, given any input v_0, there is some output w_0 that is not refused. This result relies on the fact that operations of M do not block execution.

Lemma 2. *Given an operation $w \longleftarrow e(v)$ of M: if tr is a non-divergent trace of M and (tr, X) is a failure of M, then for any input value v_0 there is some output value w_0 such that $e.v_0.w_0 \notin X$.*

Proof. We use the failures/divergences semantics for action systems (and thus B machines) given in [7], extended to include inputs and outputs in [3, 14].

Given (tr, X) as a failure of M, then (by definition) $\neg[tr](\bigvee_{e \in X} g_e)$ (where tr is considered as the sequential composition of the operations listed in it, and $g_e = \neg[e]false$ expresses that e is enabled.

Given $e.v_0.w_0$, the guard $g_{e.v_0.w_0}$ is calculated for the operation e with input v_0 and output coerced to w_0:

$$\begin{aligned} g_{e.v_0.w_0} &= \neg[w \longleftarrow e(v_0)][w = w_0]false \\ &= \neg[w \longleftarrow e(v_0)](w \neq w_0) \end{aligned}$$

Now assume for a contradiction that $e.v_0.w_0 \in X$ for all possible values of w_0 (of output type T, say), then we have

$$\begin{aligned} \neg[tr](\bigvee_{e.v_0.w_0 \in X} g_{e.v_0.w_0}) &\iff \neg[tr](\neg(\bigwedge_{w_0 \in T} \neg g_{e.v_0.w_0})) \\ &\iff \neg[tr](\neg(\bigwedge_{w_0 \in T} [w \longleftarrow e(v_0)](w \neq w_0))) \\ &\iff \neg[tr](\neg[w \longleftarrow e(v_0)](\bigwedge_{w_0 \in T} (w \neq w_0))) \\ &\iff \neg[tr](\neg[w \longleftarrow e(v_0)]\, false) \qquad (\text{since } w \in T) \end{aligned}$$

However, operation e does not block, which means that $\neg[w \longleftarrow e(v_0)]false$ is true, and so we obtain $\neg[tr]true$. On the other hand, however, tr is a non-divergent trace of M, which means that $[tr]true$, yielding a contradiction, and the result follows.

The next lemma states that if a controller process can refuse some output w from a B machine on a machine channel e (with input v), then it can refuse all possible outputs. It uses the notation $\{|\ CV\ |\} = \{c.v.w \mid c.v \in CV \wedge c.v.w \in \Sigma\}$.

Lemma 3. *Given a CSP controller P and one of its failures (tr, X), if CV is a set of machine channels e and input values v such that $\forall\, e.v \in CV\ .\ \exists\, w\ .\ e.v.w \in X$, then $(tr, X \cup \{|\ CV\ |\}) \in \mathcal{F}_{SF}\ [\![P]\!]$.*

Proof. This follows by structural induction on the clauses of the controller. The proof requires consideration of the failures semantics of each language construct, and makes use of the fact that none of them enable a controller to partially block output from a B machine (i.e. no guards on such outputs).

Proof of Theorem 2. We prove the contrapositive: that if there is a deadlock (tr, Σ) of the entire system $\|_i (P_i \parallel M_i)$ then it must also be a deadlock of $\|_i P_i$. The entire system is divergence-free by Lemma 1, so there is a deadlock in the failures/divergences semantics of the system if and only if there is a deadlock in its stable failures semantics. We consider the stable failures semantics.

Consider $(tr, \Sigma) \in \mathcal{F}_{SF}\ [\![\ \|_i (P_i \parallel M_i)]\!]$. Then there must be refusal sets $X_{P_i} \subseteq \alpha(P_i)$, $X_{M_i} \subseteq \alpha(M_i)$, for each i, such that $\bigcup_i (X_{P_i} \cup X_{M_i}) = \Sigma$, and such that for each i

- $(tr \upharpoonright \alpha(P_i), X_{P_i}) \in \mathcal{F}_{SF}\ [\![P_i]\!]$
- $(tr \upharpoonright \alpha(M_i), X_{M_i}) \in \mathcal{F}_{SF}\ [\![M_i]\!]$

Consider M_i. Define CV to be the set containing all channels $c.v$ made up of an operation name c and an input value v. By Lemma 2, for each $c.v \in CV$ there is some value w such that $c.v.w \notin X_{M_i}$. Thus for each $c.v \in CV$ there is some w such that $c.v.w \in X_{P_i}$, (since $c.v.w$ does not appear in the alphabets of any other machines). By Lemma 3 it follows that $(tr \upharpoonright \alpha(P_i), X_{P_i} \cup \{|\ CV\ |\}) \in \mathcal{F}_{SF}\ [\![P_i]\!]$. But $\alpha(M_i) \subseteq \{|\ CV\ |\}$, and so $X_{M_i} \subseteq \{|\ CV\ |\}$. Thus $(tr \upharpoonright \alpha(P_i), X_{P_i} \cup X_{M_i}) \in \mathcal{F}_{SF}\ [\![P_i]\!]$.

So it follows that

$$(tr, \bigcup_i (X_{P_i} \cup X_{M_i})) \in \mathcal{F}_{SF}\ [\![\ \|_i P_i]\!]$$

i.e. $(tr, \Sigma) \in \mathcal{F}_{SF}\ [\![\ \|_i P_i]\!]$. This means that $\|_i P_i$ has a deadlock in the stable failures model.

Hence, if $\|_i P_i$ is deadlock-free in the stable failures model, then $\|_i (P_i \parallel M_i)$ is deadlock-free in the stable failures model and the failures/divergences model.