Combining CSP and B for Specification and Property Verification*

Michael Butler[1] and Michael Leuschel[1,2]

[1] Department of Electronics and Computer Science
University of Southampton
Highfield, Southampton, SO17 1BJ, UK
{mjb,mal}@ecs.soton.ac.uk

[2] Institut für Informatik, Heinrich-Heine Universität Düsseldorf
Universitätsstr. 1, D-40225 Düsseldorf
leuschel@cs.uni-duesseldorf.de

Abstract. ProB is a model checking tool for the B Method. In this paper we present an extension of ProB that supports checking of specifications written in a combination of CSP and B. We explain how the notations are combined semantically and give an overview of the implementation of the combination. We illustrate the benefit that appropriate use of CSP, in conjunction with our tool, gives to B developments both for specification and for verification purposes.

Keywords: B-Method, Tool Support, Model Checking, Animation, Logic Programming, Constraints.

1 Introduction

The B-method, originally devised by J.-R. Abrial [1], is a theory and methodology for formal development of computer systems. It is used by industries in a range of critical domains, most notably railway control. B is based on the notion of *abstract machine* and the notion of *refinement*. The variables of an abstract machine are typed using set theoretic constructs such as sets, relations and functions. Typically these are constructed from basic types such as integers and given types from the problem domain (e.g., *Name, User, Session*, etc). The invariant of a machine is specified using predicate logic. Operations of a machine are specified as *generalised substitutions*, which allow deterministic and nondeterministic state transitions to be specified. There are two main proof activities in B: *consistency checking*, which is used to show that the operations of a machine preserve the invariant, and *refinement checking*, which is used to show that one machine is a valid refinement of another. These activities are supported by industrial strength tools, such as Atelier-B [18] and the B-toolkit [3]. In this paper, we focus on consistency checking.

In previous work [11], we have presented the ProB animator and model checker. Based on Prolog, the ProB tool supports automated consistency checking of B machines via *model checking* [5]. For exhaustive model checking, the

* This research is being carried out as part of the EU funded research project: IST 511599 RODIN (Rigorous Open Development Environment for Complex Systems).

given sets must be restricted to small finite sets, and integer variables must be restricted to small numeric ranges. This allows the checking to traverse all the reachable states of the machine. ProB can also be used to explore the state space non-exhaustively and find potential problems. The user can set an upper bound on the number of states to be traversed or can interrupt the checking at any stage. ProB will generate and graphically display counter-examples when it discovers a violation of the invariant. ProB can also be used as an animator of a B specification. So, the model checking facilities are still useful for infinite state machines, not as a verification tool, but as a sophisticated debugging and testing tool.

In the Event B approach [2], a B machine is viewed as a reactive system that continually executes enabled operations in an interleaved fashion. This allows parallel activity to be easily modelled as an interleaving of operation executions. However, while B machines are good at modelling parallel activity, they can be less convenient at modelling sequential activity. Typically one has to introduce an abstract 'program counter' to order the execution of actions. This can be a lot less transparent than the way in which one orders action execution in process algebras such as CSP [9]. CSP provides operators such as sequential composition, choice and parallel composition of processes, as well as synchronous communication between parallel processes.

Our motivation is to use CSP and B together in a complementary way. B can be used to specify abstract state and can be used to specify operations of a system in terms of their enabling conditions and effect on the abstract state. CSP can be used to give an overall specification of the coordination of operations. To marry the two approaches, we take the view that the execution of an operation in a B machine corresponds to an event in CSP terms. Semantically we view a B machine as a process that can engage in events in the same way that a CSP process can. The meaning of a combined CSP and B specification is the parallel composition of both specifications. The B machine and the CSP process must synchronise on common events, that is, an operation can only happen in the combined system when it is allowed both by the B and the CSP. There is much existing work on combining state based approaches such as B with process algebras such as CSP and we review some of that in a later section.

In [10] we presented the CIA (CSP Interpreter and Animator) tool, a Prolog implementation of CSP. As both ProB and CIA are implemented in Prolog, we were provided with a unique opportunity to combine these two to form a tool that supports animation and model checking of specifications written in a combination of CSP and B. This paper reports on the combined tool. In Section 2 we provide an overview of the ProB and CIA tools. In Section 3 we describe how the tools are combined and what the effect of the combination is.

We envisage two main uses of the combined tool. Firstly it can be used to animate and model check specifications which are a combination of B and CSP. We illustrate this in Section 4. The second use of the tool, described in Section 5, is to analyse trace properties of a B machine. In this case the behaviour is fully specified in B, but we use CSP to specify some desirable or undesirable behaviours and use ProB to find traces of the B machine that exhibit those behaviours.

2 Background

ProB ProB [11] is an animation and model checking tool for the B method. ProB 's animation facilities allow users to gain confidence in their specifications, and unlike the animator provided by the B-Toolkit, the user does not have to guess the right values for the operation arguments or choice variables. The undecidability of animating B is overcome by restricting animation to finite sets and integer ranges, while efficiency is achieved by delaying the enumeration of variables as long as possible. ProB also contains a model checker [5] and a constraint-based checker, both of which can be used to detect various errors in B specifications.

The ProB system has been developed mainly in SICStus Prolog, with graphical user interfaces implemented in Tcl/Tk and also Java. ProB uses the JBTools package to translate abstract machine notation (AMN) [1] specifications into XML, while the Pillow package allows the conversion of XML files into a Prolog term representation. The ProB front end then postprocesses the general Prolog term tree representation of the Pillow library output into a more structured representation that serves as the input to the ProB interpreter. The ProB *interpreter* recurses through this structured representation of B machines and makes calls to the ProB *kernel*, which implements support for the basic datatypes and operations of the B-language. The ProB kernel itself is written in SICStus Prolog with co-routining (i.e., `when` declarations) and constraints (finite domain constraints using CLP(FD)). The ProB animator, and the various checking tools described below all make use of the ProB interpreter in various ways.

ProB provides two ways of systematically checking a B machine: 1. a *temporal* model checking [5] which tries to find a sequence of operations that, starting from an initial state, leads to a state which violates the invariant (or exhibits some other error, such as deadlocking, assertion violations, or abort conditions); and 2. a *constraint-based* checking, which finds a state of the machine that satisfies the invariant, but where we can apply a single operation to reach a state that violates the invariant (or again exhibits some other error). More details can be found in [11]. Recently *refinement checking* has also been added, which can be used to check refinement between two B specifications. In case refinement is violated, ProB displays a sequence of operations that can be performed by the "refinement" machine but not by the specification machine.

The CSP Interpreter and Animator CSP is a process algebra defined by Hoare [9]. The first semantics associated with CSP was a denotational semantics in terms of traces, failures and (failure and) divergences. An operational semantics has later been developed [14], which forms the basis of the interpreter and animator presented in [10]. This interpreter was also developed in SICStus

Prolog. No CLP (Constraint Logic Programming) primitives were used but co-routining (i.e., `when` declarations) were used to ensure that channel constraints are delayed until they are sufficiently instantiated to evaluate them. The implementation presented in [10] covers a large part of CSP, see Figure 1. In the light of integration with ProB we have improved the parser (which uses Prolog's Definite Clause Grammars) and we have moved much closer to the CSP-M syntax as employed by FDR [15, 7].[3]

As the CSP interpreter is also written also in SICStus Prolog, at least from a technical point of view, it is now feasible to integrate CSP and B. In the following section we describe how this was done, starting out from the theoretical underpinnings and then leading on to the practical aspects.

Operator	Syntax	Ascii Syntax
stop	$STOP$	`STOP`
skip	$SKIP$	`SKIP`
prefix	$a \rightarrow Q$	`a->P`
conditional prefix	$a?x : C \rightarrow P$	`a?x:C->P`
external choice	$P \Box Q$	`P [] Q`
internal choice	$P \sqcap Q$	`P |~| Q`
interleaving	$P \vert\vert\vert Q$	`P ||| Q`
parallel composition	$P \llbracket A \rrbracket Q$	`P [| A |] Q`
sequential composition	$P; Q$	`P ; Q`
hiding	$P \backslash A$	`P \ A`
renaming	$P[R]$	`P [[ R ]]`
timeout	$P \rhd Q$	`P [> Q`
interrupt	$P \triangle_i Q$	`P /\ Q`
if then else	$if\ C\ then\ P\ else\ Q$	`if C then P else Q`
let expressions	$let\ v = e\ in\ P$	`let V=E in P`
agent definition	$A = P$	`A = P;`

Fig. 1. Summary of syntax of CSP

3 Combining B and CSP

In our work we have adopted and developed the approach of integration depicted in Figure 2. (How this compares to earlier work is discussed later in the paper.) In essence, the B and CSP specifications are composed in parallel. The B operations must synchronize with channel events of the CSP specification having the same

[3] But there are still a few differences and extra features. For example, multiple process definitions are allowed and treated like an external choice and process definitions can be terminated by a double semicolon to ease error recovery during parsing. However, variable names still have to start with an uppercase letter or an underscore and channel declarations are ignored.

name as the B operation. Channel events of the CSP which have no counterpart in the B (such as channel D in Figure 2) can occur independently, while B operations that have no CSP counterpart are prevented from being executed. Below we present more formally how this synchronization is achieved, starting out from the state information of a combined B/CSP specification and then progressing on to how to formally perform the synchronization.

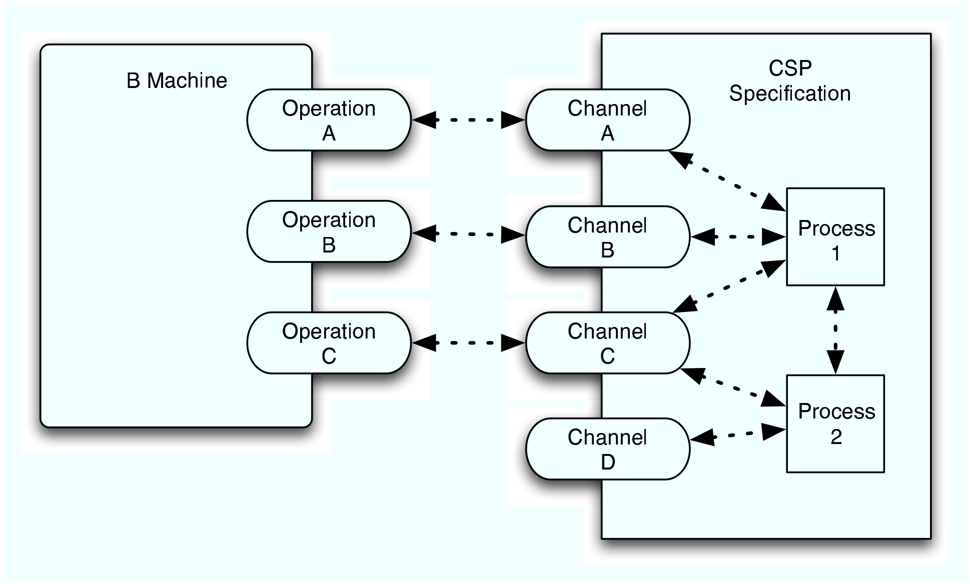

Fig. 2. Illustrating the synchronisation of B and CSP specifications

Combining State Information The state of a B machine is a mapping from variables to values, while the state of a CSP process is a syntactic process expression. So, for example, the state of the simple B machine in Figure 3 immediately after executing $Set(cc)$ would be represented as $\{xx \mapsto cc\}$[4], while the state of the simple CSP specification in Figure 4 would be $Get.cc \rightarrow MAIN$. A state of a combined B/CSP specification is thus simply a pair, whose first component is a B state and second component a CSP process expression. For example, the state of the combination of Figures 3 and 4 immediately after executing $Set(cc)$ would be $(\{xx \mapsto cc\}, Get.cc \rightarrow MAIN)$.

Mapping Operations to Channels The approach we have chosen is to translate every argument and return value of a B operation into a separate data value

[4] The actual Prolog representation is `[bind(xx,fd(3,'AA'))]`.

```
MACHINE  $Simple$           OPERATIONS
SETS
   $AA = \{aa, bb, cc\}$      $Set(newval) \hat{=}$
VARIABLES                     PRE  $newval \in AA$
   $xx$                        THEN  $xx := newval$
INVARIANT                     END;
   $xx \in AA$               $res \longleftarrow Get =$
INITIALISATION                BEGIN        $res := xx$
   $xx := aa$                  END
```

Fig. 3. Simple B machine

$$Set?Val \rightarrow Get!Val \rightarrow MAIN$$

Fig. 4. Simple CSP Specification

of a CSP channel. To ease the writing of succinct CSP specifications, we do not require the CSP to provide all channel values. If channel values are missing any B value is allowed for synchronization on that argument.

For a B operation of the form $X \longleftarrow op(Y) \hat{=} S$, we refer to $a \longleftarrow op(b)$ as an operation call. We first define a function *channel* which maps B operation calls to possible CSP channel events. Let *op* be an operation of a B machine taking $n \geq 0$ arguments and returning $m \geq 0$ values, and let $a_1, \ldots, a_n$ be arguments to that operation and let $r_1, \ldots, r_m$ be return values. We then define

$$\begin{aligned} channel(r_1, \ldots, r_m \leftarrow op(a_1, \ldots, a_n)) \ &= \ \{\ op.a_1. \ldots .a_k \mid 0 \leq k \leq n\} \ \cup \\ &\quad\ \{\ op.a_1. \ldots .a_n.r_1. \ldots .r_k \mid 1 \leq k \leq m\} \end{aligned}$$

(If m is 0 we take the liberty of not writing the result arrow "$\leftarrow$".)

For example for the B machine in Figure 3, we have

$$\begin{aligned} channel(Set(aa)) \ &= \ \{Set, Set.aa\} \\ channel(aa \leftarrow Get) \ &= \ \{Get, Get.aa\} \end{aligned}$$

Intuitively, this means that a channel event *Set* will synchronise with all possible executions of the B operation *Set*, whereas *Set.aa* will only synchronise with the execution of *Set* for the particular argument *aa*.

Deriving an Operational Semantics We suppose that the B operational semantics is given by a ternary relation $\rightarrow$ (in practice computed by ProB), where $\sigma \rightarrow_o \sigma'$ with $o = r_1, \ldots, r_m \leftarrow op(a_1, \ldots, a_n)$ means that in the state σ of a B machine we can execute the operation *op* with arguments $a_1, \ldots, a_k$ giving the return values $r_1, \ldots, r_m$ and producing the new state σ'.

The CSP operational semantics is given by a similar relation $\rightarrow$, where $P \rightarrow_{ch.a_1.\ldots.a_n} P'$ denotes the fact that the process expression P can produce the channel event $ch.a_1. \ldots .a_n$ and evolve into the new process expression P'.

We can now define our new operational semantics of a combined B and CSP specification by $(\sigma, P) \rightarrow_A (\sigma', P')$ iff $\sigma \rightarrow_O \sigma'$ and $P \rightarrow_A P'$ and $A \in \mathit{channel}(O)$.

Computing the Operational Semantics The question now is: how can we compute $(\sigma, P) \rightarrow_A (\sigma', P')$ in practice? The first part, $\sigma \rightarrow_O \sigma'$, is computed by ProB, while $P \rightarrow_A P'$ is computed by the CSP interpreter. The remaining final part, checking $A \in \mathit{channel}(O)$ has been implemented by unifying the B operation arguments with the CSP channel values (those provided). So, synchronisation is achieved by Prolog unification. This means we have a very flexible way of combining CSP and B as information can flow in both directions. In other words, the CSP can drive the B or vice-versa or a combination thereof. The use of co-routining in both ProB and the CSP interpreter not only makes this kind of synchronization possible but also efficient. Indeed, both the B and CSP parts can provide concrete date values, and as soon as those are available the co-routining mechanism will trigger the relevant tests in either the B or the CSP part (or both). If any of those tests fail the search space is immediately pruned, resulting in a (possibly considerable) efficiency gain, when compared to computing the B and CSP operational semantics in isolation.[5]

Observe that in this translation, no distinction is made between arguments and return values. Indeed, ProB itself makes little distinction between arguments and return values (the only difference is that it is easier to extract typing for arguments). This allows for a very flexible way of synchronising, e.g., giving the CSP the option of imposing return values or just retrieving them.

The Implementation The above described combination of B and CSP has been integrated into the latest release of ProB. Many of ProB's features specific to B continue to work for combined B/CSP specifications: backtrackable automatic animation, graphical visualization possibly with optional state space reduction [12], temporal model checking with detection of invariant, assertion violations or deadlocks, refinement checking, and many more.

Figure 5 shows the state space, as visualized by the new version of ProB, for the combination of the B machine from Figure 3 and the CSP specification from Figure 4. The figure clearly shows how the CSP has imposed that every Set operation is followed by a Get operation. The CSP also imposes that the Get operation must return the same value as was given to the Set operation. Hence, the absence of deadlocks in Figure 5 (formally verified by ProB) can be viewed as proving a temporal property of the B machine: whenever one does a Set operation with argument x one can perform a Get operation and the result is equal to x.[6] We will return to this usage of combining CSP and B in Section 5.

[5] In the worst case if both B and CSP wait for each other to provide concrete data values the ProB enumeration of the B datatypes will be triggered and drive the interpreter.

[6] In CTL one could write $\forall x.(AG\ \mathit{Set}(x) \Rightarrow X(x \leftarrow \mathit{Get}))$.

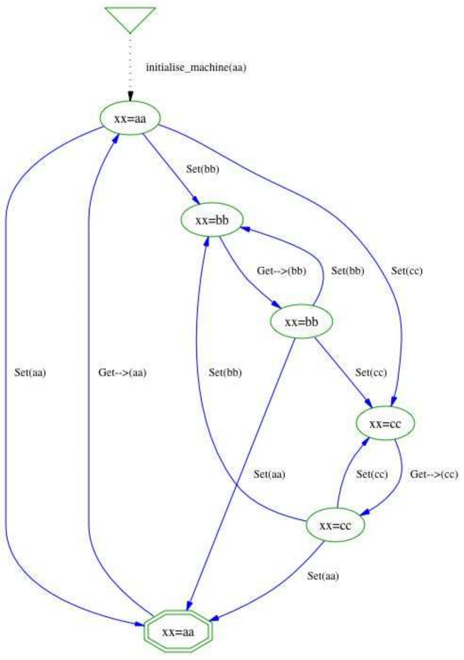

Fig. 5. The state space of combination of the two simple B and CSP specifications

To conclude this section, let us use the same B machine from Figure 3 but use the following CSP specification:

$$MAIN = Set \rightarrow Cst \qquad\qquad Cst = Get \rightarrow Cst$$

Here, we have used the CSP to ensure that the B machine variable can only be assigned once, and that its value can only be read after it has been assigned. This is a simple illustration of how one can use the combination of B and CSP for specification purposes, and the state space computed by ProB can be found in Figure 6. In the next section we will illustrate this usage on a more interesting example.

4 Specifying using B and CSP

In this section we illustrate the use of a combination of B and CSP to specify a system. The example we use to illustrate this concerns a service for distributing tokens to customers via offices and is based on [8]. The B part of our specification

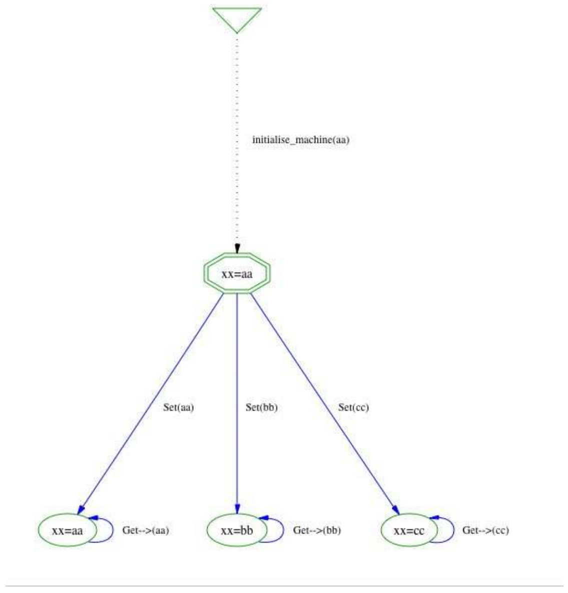

Fig. 6. The state space of *Simple* when using another CSP specification

models a database mapping customers to the number of available tokens (Figure 7). It provides operations for creating and deleting customers which add or remove mappings for a customer to or from the database. There are operations for allocating a token to a customer as well as operations for requesting tokens and collecting tokens. Requesting tokens has no effect on the database. If there is more than one token available for a customer, the number of tokens returned is nondeterministically chosen to be less than or equal to the number of tokens available for that customer.

The finiteness of the sets $OFFICE$ and $CUST$ in Figure 7 is required for exhaustive model checking. Finiteness is also impose by restricting the maximum number of tokens allocated to a customer using the constant mx. The $AllocToken$ operation is guarded to ensure that this allocation is never exceeded.

We wish to impose a certain coordination protocol on the operations of the system in Figure 7. Operations such as $CollectToken$ and $AllocToken$ should only be available after a customer has been added to the system. Furthermore, before a customer can collect tokens, they must first request those tokens at an office. This coordination is described by the CSP process $MAIN$ of Figure 7. This process consists of three parallel instances of the $Cust$ process, one for each cus-

```
MACHINE $Tokens$
SETS
  $OFFICE = \{o1, o2\}$;
  $CUST = \{c1, c2, c3\}$
CONSTANTS $mx$
PROPERTIES $mx \in \mathbb{N} \land mx = 3$
VARIABLES $tokens$
INVARIANT $tokens \in CUST \mathrel{+\!\!\to} (0..mx)$

INITIALISATION $tokens := \{\}$

OPERATIONS

$AddCust(cc) \mathrel{\hat=}$
  PRE $cc \in CUST \land cc \notin dom(tokens)$
  THEN $tokens := tokens \cup \{cc \mapsto 0\}$
  END;

$RemCust(cc) =$
  PRE $cc \in CUST$
  THEN $tokens := \{cc\} \lhd tokens$
  END;

$AllocToken(cc) =$
  PRE $cc \in CUST \land cc \in dom(tokens)$
  SELECT $tokens(cc) < mx$ THEN
    $tokens(cc) := tokens(cc) + 1$ END
  END;

$ReqToken(cc, pp) =$
  PRE $cc \in CUST \land pp \in OFFICE$
  THEN $skip$
  END;

$toks \longleftarrow CollectToken(cc, pp) =$
  PRE $cc \in CUST \land pp \in OFFICE \land$
      $cc \in dom(tokens)$
  THEN
    IF $tokens(cc) = 0$
    THEN $toks := 0$
    ELSE
      ANY $nn$ WHERE $nn : \mathbb{N} \land$
        $1 \leq nn \land nn \leq tokens(cc)$
      THEN $toks := nn \mid\mid$
        $tokens(cc) := tokens(cc) - nn$
      END END END
```

Fig. 7. Tokens B machine

tomer. In a *Cust* process, *AddCust* is the only operation available initially. Once *AddCust* has been performed, allocation and collection of tokens can proceed in parallel, modelled by the process $(Collection(C)[|RemCust|]Allocation(C))$. Collection and allocation synchronise on the *RemCust* event because both are terminated by this event. Collection of tokens by a customer is intended to take place at offices to which customers have access. Before customers can collect tokens from an office, they must first request tokens at that office via a *ReqToken* operation. Only then can they collect some (or all) of the tokens available for them. The definition of *Collection* also ensures that a customer cannot be removed in between requesting some tokens and collecting those tokens.

The overall behaviour of the service is determined by the parallel composition of the B and CSP parts. In this case, the CSP specification ensures that the *AddCust* operation must be invoked before any of the other operations are allowed, and that tokens must be requested before they can be collected. The ProB tool allows the combined specification to be animated so that the overall behaviour can be explored interactively.

Now consider the preconditions of the operations of Figure 7. The *AddCust* operation has $cc \notin dom(tokens)$ as a precondition, while the *AllocToken* and *CollectToken* operations have $cc \in dom(tokens)$ as a precondition. The preconditions represent assumptions about the conditions under which these operations will be invoked but are not enforced by the B machine on its own. Normally,

$$\begin{aligned}
MAIN &= Cust(c1)\ \ |||\ \ Cust(c2)\ \ |||\ \ Cust(c3) \\[1em]
Cust(C) &= AddCust.C \rightarrow (Collection(C)[|RemCust|]Allocation(C))\ ;\ Cust(C) \\[1em]
Collection(C) &= (\ ReqToken.C?O \rightarrow CollectToken.C.O \rightarrow Collection(C) \\
&\quad\ \square\ RemCust.C \rightarrow SKIP) \\[1em]
Allocation(C) &= (\ AllocToken.C \rightarrow Allocation(C) \\
&\quad\ \square\ RemCust.C \rightarrow SKIP)
\end{aligned}$$

Fig. 8. Tokens CSP equations

when checking the consistency of a B machine using ProB, operation preconditions are used to restrict the reachable states by treating them in exactly the same way as operation guards. This form of checking detects no errors in the machine of Figure 7. An alternative form of checking can be applied in ProB which treats a violation of a precondition as an error. That is, an error is raised if a machine can reach a state which violates an operation precondition. With this second form of model checking, when the machine of Figure 7 is checked, an error is detected straightaway because the initial state violates the preconditions of *AllocToken* and *CollectToken*. However, when this form of checking is applied to the combined B and CSP specification, no violation of preconditions is detected by ProB. This is because the CSP enforces an order on the invocation of the operations which guarantees that the preconditions are always satisfied.

5 Verifying properties of B machines using CSP

In the previous section, we illustrated how a system could be specified as a combination of CSP and B. In this section we illustrate how CSP specifications can be used to analyse trace properties of specifications written purely in B. With this approach we use CSP to specify some desirable or undesirable behaviours and use ProB to find traces of the B machine that exhibit those behaviours. To specify a desirable property, we use a special CSP process called *GOAL*. A desirable trace is one that leads to the *GOAL* process. An undesirable trace is one that leads to the *ERROR* process.

To illustrate the use of *GOAL* and *ERROR*, we consider a simple mobile agent system. Once agents have been created they can have a location or be in transit between locations. When an agent is at some location, it can send and receive messages to and from other agents. Messages can be sent to agents even if they are in transit in which case the messages can be received when the receiving agent reaches a location. The simple agent system is specified by the B machine of Figure 9. In this specification, *agents* represents the set of created agents, $msgs(a)$ represents the set of messages waiting to be received by agent a, and $loc(a)$ represents the location of agent a. If a is in $agents$ but not in the domain of loc, then a is in transit.

```
MACHINE  $MobileAgents$
SETS    $MSG = \{m1, m2\}$;
  $AGENT = \{a1, a2\}$;
  $LOC = \{l1, l2\}$
VARIABLES  $agents, loc, msgs$
INVARIANT
  $agents \in \mathbb{P}(AGENT) \wedge$
  $msgs \in agents \rightarrow \mathbb{P}(MSG) \wedge$
  $loc \in agents \mathrel{+\!\!\!\rightarrow} LOC$
INITIALISATION
  $agents := \{\} \mid\mid msgs := \{\} \mid\mid loc := \{\}$
OPERATIONS

$Create(aa) \hat{=}$
  PRE
    $aa \in AGENT \setminus agents$
  THEN
    $agents := agents \cup \{aa\} \mid\mid$
    $msgs := msgs \cup \{aa \mapsto \{\}\}$
  END;

$Arrive(aa, ll) \hat{=}$
  PRE
    $aa \in agents \setminus dom(loc) \wedge$
    $ll \in LOC$
  THEN
    $loc(aa) := ll$
  END;

$Depart(aa, ll) \hat{=}$
  PRE
    $aa : agents \wedge ll : LOC \wedge$
    $(aa| \mapsto ll) \in loc$
  THEN
    $loc := \{aa\} \lhd loc$
  END;

$Send(aa, bb, ll, mm) \hat{=}$
  PRE
    $aa \in agents \wedge bb \in agents \wedge$
    $mm \in MSG \wedge ll \in LOC \wedge$
    $aa \neq bb \wedge (aa \mapsto ll) \in loc$
  THEN
    $msgs(bb) := msgs(bb) \cup \{mm\}$
  END;

$mm \longleftarrow Receive(bb, ll) \hat{=}$
  PRE
    $bb \in agents \wedge ll \in LOC \wedge (bb \mapsto ll) \in loc$
  THEN
    ANY  $m1$ $WHERE$
      $m1 \in MSG \wedge m1 \in msgs(bb)$
    THEN
      $msgs(bb) := msgs(bb) - m1 \mid\mid$
      $mm := m1$
    END
  END
```

Fig. 9. Mobile agents B machine

A desirable property of the agent system is that it is possible for an agent to receive a message since this is an important service for agents. Clearly some sequence of operations must happen before an agent can receive a message. There is a danger that our specification of the operations is too restrictive so that a trace leading to receipt of a message would not be possible. Figure 10 contains a CSP process which leads to the $GOAL$ process when a *Receive* event is executed.

We do not want the CSP process to place any constraints on the *Create*, *Send*, *Arrive* or *Depart* operations. To achieve this we use the special RUN process. RUN takes a list of events and continually iterates over the choice of

$$\begin{aligned} MAIN &= Test1 \;|||\; RUN[Create, Send, Arrive, Depart] \\ Test1 &= Receive \rightarrow GOAL \end{aligned}$$

Fig. 10. Goal test for agents

those events. For example, we have

$$RUN[A, B] \;=\; (\; A \rightarrow RUN[A, B] \quad \Box \quad B \rightarrow RUN[A, B] \;)$$

We interleave the $Test1$ process with $RUN[Create, Send, Arrive, Depart]$. If RUN was not interleaved with the test process, then the $Create$, $Send$, $Arrive$ and $Depart$ operations could never take place in the combined system.

Figure 12 is generated by ProB and it illustrates a trace of events and corresponding machine states which lead to a message being received by an agent. All of the events leading to the receipt are required because both agents need to exist before one can send a message to the other and an agent needs to be at a location in order to send or receive. As well as the event trace, the diagram allows us to see the evolution of the state of the B machine. This is useful for helping to validate the specification. To find the desirable trace, ProB checks the parallel composition of the B machine and the CSP process, attempting to find a trace leading to the $GOAL$ process. In this case it finds the trace illustrated in Figure 12.

$$\begin{aligned} MAIN &= TEST2 \;|||\; RUN[Create, Arrive, Depart, Send?A?B?L.m2, Receive] \\ TEST2 &= Receive?A?L.m1 \rightarrow ERROR \end{aligned}$$

Fig. 11. Error test for agents

An undesirable behaviour of agent system would be that an agent receives a message without the message having been sent to the agent. This behaviour is encoded in the CSP process of Figure 11. The main constraint imposed by this CSP process is that the $Send$ operation is prevented from sending message $m1$. An error arises when message $m1$ is received by an agent. Receipt of message $m1$ represents an error since it could not have been preceded by a send of $m1$ because of the constraint on sending. In this case, ProB performs an exhaustive search but fails to find a trace leading to the $ERROR$ process. This is as expected since the $Receive$ operation requires a recipient to have some message in their message set, and messages only get added to a message set through the $Send$ operation.

There is a very significant difference between a *GOAL* test and an *ERROR* test. Success of a *GOAL* test gives us a single trace leading to the goal. It tells us nothing about all possible behaviours of the system. Nonetheless the existence of the goal trace can increase our confidence in the validity of the B model and can be used to provide guided automatic animation of a B machine. We deem an *ERROR* test to be successful when ProB finds no trace leading to the error process through exhaustive search. In the case of the error test for the agent system, the only constraint the CSP process places on the B machine is to prevent sending of message $m1$. The absence of any error traces means there is no trace of the agent system which contains a receipt of $m1$ but does not contain a send of $m1$.

6 Related Work and Conclusion

Our combined CSP and B tool is most strongly related to the csp2B tool [4] and the CSP‖B approach [16]. The csp2B tool allows specifications to be written in a combination of CSP and B by compiling the CSP to a pure B representation which can be analysed by a standard B tool. The CSP support by csp2B is more restricted that that supported by ProB: csp2B does not support internal choice and allows parallel composition only at the outermost level unlike the arbitrary combination of CSP operators supported by ProB. The work of [16] is focused on a style of combining CSP and B where the B machines are passive and all the coordination is provided by the CSP. This means the operations of their B machines cannot be guarded though they can have preconditions. They have developed compositional rules for proving that CSP controllers do not lead to violation of operation preconditions.

There has been much work on combining CSP with Z and Object-Z, including [6] and [17]. Like our approach, these treat Z specifications as CSP processes and model the composition of the CSP and Z parts as parallel composition. The work described in [13] describes an approach to translating Z to CSP so that CSP-Z specifications can be model checked. This translation is not automated though. The Circus language is a rich combination of Z and CSP allowing Z to be easily embedded in CSP specifications and providing refinement rules for development [19]. We are not aware of any tools that allow for model checking of Z and CSP specifications directly.

The combined model checker for CSP and B as an enhancement of the existing ProB checker allowing for automated consistency checking of specifications written in a combination of CSP and B. We have shown how ProB can now be used to automatically check consistency between B and CSP specifications (i.e., checking that no B preconditions are ever violated). We have also shown how ProB can be used to check whether a pure B specification satisfies traces properties expressed in CSP. This form of checking serves to increase our confidence in the validity of B models.

ProB also supports refinement checking between B models and between combinations of CSP and B. Further work is required to enhance the scalability of the model checking approach, especially for refinement checking (although some quite large, realistic specifications have already been successfully verified). Our view is that ProB is a valuable complement to the usual theorem prover based development in B. Wherever possible there is value in applying model checking to a size-restricted version of a B model before attempting semi-automatic deductive proof.

References

1. J.-R. Abrial. *The B-Book.* Cambridge University Press, 1996.
2. J.-R. Abrial and L. Mussat. Introducing dynamic constraints in B. In D. Bert, editor, *Second International B Conference*, April 1998.
3. B-Core (UK) Limited, Oxon, UK. *B-Toolkit.*, 1999. Available at www.b-core.com
4. M. J. Butler. csp2B: A Practical Approach to Combining CSP and B. *Formal Asp. Comput.*, 12(3):182–198, 2000.
5. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 1999.
6. C. Fischer. CSP-OZ: A combination of Object-Z and CSP. *Formal Methods for Open Object-Based Distributed Systems (FMOODS'97)*, pages 423–438. Chapman & Hall, 1997.
7. Formal Systems (Europe) Ltd. *FDR2 User Manual.*
8. P. Hartel, M. Butler, A. Currie, P. Henderson, M. Leuschel, A. Martin, A. Smith, U. Ultes-Nitsche, and B. Walters. Questions and answers about ten formal methods. In *Proc. 4th Int. Workshop on Formal Methods for Industrial Critical Systems*, Trento, Italy, Jul 1999.
9. C.A.R. Hoare. *Communicating Sequential Processes.* Prentice–Hall, 1985.
10. M. Leuschel. Design and implementation of the high-level specification language CSP(LP) in Prolog. *Proceedings of PADL'01*, LNCS 1990, pages 14–28. Springer-Verlag, March 2001.
11. M. Leuschel and M. Butler. ProB: A Model Checker for B. *Proceedings FME 2003, Pisa, Italy*, LNCS 2805, pages 855–874. Springer, 2003.
12. M. Leuschel and E. Turner. Visualizing larger states spaces in ProB. In *Proceedings ZB'2005*, LNCS. Springer-Verlag, April 2005. To appear.
13. A. Mota and A. Sampaio. Model-checking CSP-Z: strategy, tool support and industrial application. *Sci. Comput. Program.*, 40(1):59–96, 2001.
14. A. Roscoe. *The Theory and Practice of Concurrency.* Prentice–Hall, 1998.
15. J. B. Scattergood. *Tools for CSP and Timed-CSP.* PhD thesis, Oxford University, 1997.
16. S. Schneider and H. Treharne. Verifying controlled components. *Proceedings Integrated Formal Methods, IFM 2004, Canterbury, UK*, LNCS 2999, pages 87–107. Springer, 2004.
17. G. Smith. A semantic integration of Object-Z and CSP for the specification of concurrent systems. *Proceedings FME '97*, LNCS 1313, pages 62–81. Springer, 1997.
18. Steria, Aix-en-Provence, France. *Atelier B, User and Reference Manuals*, 1996. Available at http://www.atelierb.societe.com/index_uk.html.
19. J. Woodcock and A. Cavalcanti. The semantics of Circus. *Proceedings ZB 2002, Grenoble, France*, LNCS 2272, pages 184–203. Springer, 2002.

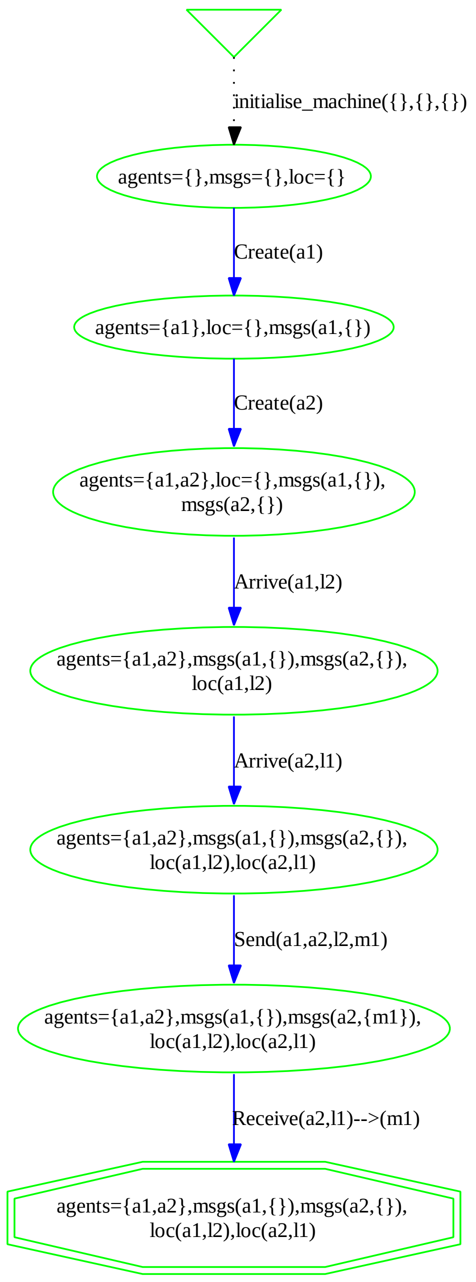
initialise_machine({},{},{})
agents={},msgs={},loc={}
Create(a1)
agents={a1},loc={},msgs(a1,{})
Create(a2)
agents={a1,a2},loc={},msgs(a1,{}), msgs(a2,{})
Arrive(a1,l2)
agents={a1,a2},msgs(a1,{}),msgs(a2,{}), loc(a1,l2)
Arrive(a2,l1)
agents={a1,a2},msgs(a1,{}),msgs(a2,{}), loc(a1,l2),loc(a2,l1)
Send(a1,a2,l2,m1)
agents={a1,a2},msgs(a1,{}),msgs(a2,{m1}), loc(a1,l2),loc(a2,l1)
Receive(a2,l1)-->(m1)
agents={a1,a2},msgs(a1,{}),msgs(a2,{}), loc(a1,l2),loc(a2,l1)

Fig. 12. Trace leading to *GOAL* process as displayed by ProB