csp2B: A Practical Approach To Combining CSP and B

Michael Butler
Department of Electronics & Computer Science
University of Southampton
Highfield
Southampton SO17 1BJ
United Kingdom
`M.J.Butler@ecs.soton.ac.uk`
`www.ecs.soton.ac.uk/~mjb`

February 14, 2000

Abstract

This paper describes the tool csp2B which provides a means of combining CSP-like descriptions with standard B specifications. The notation of CSP provides a convenient way of describing the order in which the operations of a B machine may occur. The function of the tool is to convert CSP-like specifications into standard machine-readable B specifications which means that they may be animated and appropriate proof obligations may be generated. Use of csp2B means that abstract specifications and refinements may be specified purely using CSP or using a combination of CSP and B. The translation is justified in terms of an operational semantics.

1 Introduction

In the B method [1], a system is specified as an abstract machine consisting of some state and some operations acting on that state. Originally B was intended for the development of non-distributed systems. Influenced by Action Systems [3], recent work has shown how B may be used in the development of distributed systems [2, 5, 7]. In these approaches, the state of a machine may be used to model the global state of a distributed system and its operations may represent events that change the state of the system. Refinement in this approach involves partitioning the global state amongst the nodes of the system to localise events. Events are guarded by conditions on the state and may only be executed when their guard is enabled.

However, while B is suitable for modelling distributed activity in terms of events, it is weaker 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] and CCS [10].

The csp2B tool converts CSP-like descriptions of system behaviour into standard machine-readable B specifications. The resulting B specifications can be input to a tool such as *Atelier B* from Steria and *The B-Toolkit* from B-Core which means that they may be animated and appropriate proof obligations may be generated.

The tool supports a CSP-like process notation containing prefixing ($\rightarrow$), choice ($\Box$) and the deadlocked process *STOP*. It does not support an internal nondeterminism operator. Parallel composition is supported but only at the outermost level, that is, a system can be described using a parallel composition of purely sequential processes. Interleaving of multiple instances of similar processes is also supported.

Given a CSP description of a system, the tool generates a B machine containing variables corresponding to the implicit states of the CSP processes, i.e., abstract program counters. For each event in the alphabet of the CSP description, a B operation is generated which is guarded appropriately and which updates the abstract program counters appropriately. It is possible to declare that a CSP description constrains the behaviour of a standard existing B machine, in which case, a guarded call to the corresponding operation in that existing machine is embedded in each generated operation.

We take an operational approach to the semantics of the CSP and B combination and show that the composition of a CSP process with a B machine is compositional with respect to refinement.

Section 2 gives an overview of the tool and how it may be used, while Section 3 discusses the semantics of the CSP notation used and how it relates to B.

The csp2B tool itself may be downloaded from `http://www.ecs.soton.ac.uk/~mjb/csp2B`.

2 Tool Overview

The csp2B tool converts CSP-like descriptions of system behaviour into B machines. CSP provides a very convenient way of specifying the order in which operations may be invoked. Consider the following CSP specification of a vending machine (written in the source notation of csp2B[1]):

```
MACHINE  VendingMachine

    ALPHABET  Coin   Tea   Coffee

    PROCESS  VM  =  AwaitCoin  WHERE
             AwaitCoin     =    Coin → DeliverDrink
          DeliverDrink     =     Tea → AwaitCoin
                              []  Coffee → AwaitCoin
    END
END .
```

This describes a machine that has three operations, *Coin*, *Tea* and *Coffee* (called the *alphabet* of the machine) whose behaviour is dictated by a CSP process *VM* that may be in one of two states *AwaitCoin* and *DeliverDrink*. *VM* specifies that, in the state *AwaitCoin*, *Coin* is the only operation that may be invoked while, in the *DeliverDrink* state, both the *Tea* and *Coffee* operations may be invoked. *VM* will initially be in the *AwaitCoin* state. *VM* is described by a mutually recursive set of equations and each recursive call on a right-hand side must be preceded by at least one event (in the terminology of CSP, each recursive call must be guarded). From the above CSP description, csp2B will generate the following B machine which contains a single variable *VM* and three operations *Coin*, *Tea*, and *Coffee*:

```
MACHINE  VendingMachine
SETS  VMState  =  { AwaitCoin, DeliverDrink }
VARIABLES  VM
INVARIANT  VM ∈ VMState
INITIALISATION  VM := AwaitCoin

OPERATIONS

   Coin   ≙   SELECT  VM = AwaitCoin  THEN  VM := DeliverDrink  END;

    Tea   ≙   SELECT  VM = DeliverDrink  THEN  VM := AwaitCoin  END;

 Coffee   ≙   SELECT  VM = DeliverDrink  THEN  VM := AwaitCoin  END

END .
```

[1]The tool supports an ascii version of CSP and the full syntax may be found in [6].

The operations of the generated machine are described using SELECT statements. These provide a means of specifying reactive systems in which operations are only enabled in certain states. A statement of the form

SELECT G THEN S END

is enabled only in those states for which the guard G is true. The generated B machine contains a 'control' variable named VM, the same as the name of the main process in the CSP description, of type $VMState$. The operations are guarded by and make assignments to this variable appropriately.

The semantics of B operations is given in terms of weakest preconditions. For statement S and postcondition Q, $[S]Q$ represents the weakest precondition under which S is guaranteed to terminate in a state satisfying Q. The guard of a B operation S is defined using $[S]$ as follows[2] [2, 11]:

$$grd(S) \;\hat{=}\; \neg\,[S]\,false \;.$$

From this it is easy to show that

$$\begin{aligned} grd(\text{ SELECT } G \text{ THEN } S \text{ END }) &= G \wedge grd(S) \\ grd(\ x := E\ ) &= true\ . \end{aligned}$$

2.1 Nested Prefixing

Nested prefixing in a CSP description is supported by the tool. For example, the vending machine could have been specified using a single equation:

$$AwaitCoin \;=\; Coin \rightarrow (Tea \rightarrow AwaitCoin \;\;[]\;\; Coffee \rightarrow AwaitCoin)\ .$$

In this case, the process enters an implicit unnamed state immediately after the $Coin$ event. The tool will generate a fresh name for each such implicit state in the CSP description. For the above example, csp2B will generate a fresh name for this state based on the name on the left hand side of the equation as follows:

SETS $VMState \;=\; \{\ AwaitCoin,\ AwaitCoin\_1\ \}$

$Coin \;\hat{=}\;$ SELECT $VM = AwaitCoin$ THEN $VM := AwaitCoin\_1$ END .

2.2 Parallel Processes

It is possible to have more than one process description in a single CSP specification. For example, if for some reason we wanted the vending machine to always alternate between delivering tea and coffee, we could add a process, in this case called $Alternate$, as follows:

```
MACHINE  VendingMachine

    ALPHABET  Coin   Tea   Coffee

    PROCESS  VM  =  AwaitCoin  WHERE   ... END

    PROCESS  Alternate  =  Alt
    CONSTRAINS  Tea   Coffee   WHERE
        Alt  =  Coffee → Tea → Alt
    END
END .
```

[2] $[S]\ false$ represents those initial states in which A could establish any postcondition, i.e., behave miraculously. An action is said to be enabled when it cannot behave miraculously, i.e., when $\neg\ [S]\ false$ holds.

The (optional) *CONSTRAINS* clause in the *Alternate* process signifies that this process description only constrains the *Tea* and *Coffee* operations and places no constraint on when the *Coin* operation may occur.

In the generated machine, the operations constrained by more than one process will be composed of several parallel SELECT statements. For example, the *Coffee* action will be as follows:

```
Coffee  $\hat{=}$
    SELECT  $VM = DeliverDrink$  THEN  $VM := AwaitCoin$ END
    $||$
    SELECT  $Alternate = Alt$  THEN  $Alternate := Alt\_1$ END .
```

The guard of a parallel statement satisfies the following [1]:

$$grd(S \mid\mid T) \quad = \quad trm(S) \wedge trm(T) \ \Rightarrow\ grd(S) \wedge grd(T)\ .$$

Here, $trm(S)$ is the termination condition of S. The tool always generates statements from CSP descriptions whose termination condition is always true (such as the SELECT statements for *Coffee* above). In that case, $grd(S \mid\mid T) \ = \ grd(S) \wedge grd(T)$. This means that events common to several processes will only be enabled when each of those processes is willing to engage in that event. This corresponds to the CSP notion of parallel composition (see Section 3.4).

2.3 Parameterised Events and Indexing

In the manner of channels in CSP, events may be parameterised by input parameters ($Ev?x$) or output parameters ($Ev!y$). When translated into B, these parameters will correspond to the input and output parameters of an operation. Also, processes may be indexed by parameters and these index parameters become state variables in the generated B machine. As an example of each of these, consider the following two process equations:

$$Idle \ = \ In?f \rightarrow Remember(f)$$
$$Remember(f) \ = \ Out!f \rightarrow Idle\ .$$

In the terminology of CSP, *In* and *Out* are input and output channels respectively. Process *Idle* inputs a value f on channel *In* and then behaves as *Remember* indexed by f. $Remember(f)$ outputs the index value on channel *Out* and then behaves as *Idle*.

The input parameter acts as a bound variable and its scope is the syntactic process term which the event prefixes. The output value may be defined by any B expression. The input and output parameters must be declared in the alphabet of a CSP specification in the form:

$$(y1, y2, ...) \longleftarrow OpName(x1 : T1, x2 : T2, ...)\ .$$

The types of the input parameters are needed for the generated B machine, but the output types are not needed as they are inferred by a B tool. Types are any B expression. The event in the CSP description corresponding to the above declaration is written in the form

$$OpName?x1?x2?...!y1!y2!...$$

A file transfer service that inputs files (sequences of bytes) and then outputs them may be specified as follows:

```
MACHINE  FileTransfer
SETS  Byte
DEFINITIONS  File == seq Byte
ALPHABET  Send(f : File)    f <-- Receive
    PROCESS  Copy = Idle  WHERE
        Idle = Send?f -> Remember(f)
        Remember(g : File) = Receive!g -> Idle
    END
END .
```

```
MACHINE  $FileTransfer$
SETS  $Byte$;  $CopyState = \{ Idle, Remember \}$
VARIABLES  $Copy, g$
DEFINITIONS  $File == seq(Byte)$
INVARIANT  $Copy \in CopyState \ \wedge \ g \in File$
INITIALISATION  $Copy := Idle$
OPERATIONS
    $Send(f) =$
        PRE  $f \in File$  THEN
            SELECT  $Copy = Idle$
            THEN  $Copy := Remember \ \ || \ \ g := f$ END
        END;
    $f \longleftarrow Receive =$
        SELECT  $Copy = Remember$
        THEN  $Copy := Idle \ \ || \ \ f := g$ END
END
```

Figure 1: Generated machine for *FileTransfer*.

The indexing variable g in this specification becomes a state variable of type *File* in the generated B machine shown in Figure 1. It is called g to distinguish it from the input and output parameter f of the *Send* and *Receive* operations. In the generated B machine, the *Send* operation assigns the value of its input parameter f to the variable g while the *Receive* operation reads from g. Within a PROCESS description the same indexing variable may be used in several equations and each occurrence will refer to the same variable in the generated machine.

Note that the *Send* operation in Figure 1 has a precondition which constrains the input parameters as well as a guard[3]. The precondition is required in B to determine the type of the input parameter.

Idle could also be declared using nested prefixing as follows:

$$Idle \ = \ Send?f \rightarrow Receive!f \rightarrow Idle \ .$$

In this case, as well as introducing a name for the implicit state immediately after the *Send* event, csp2B also introduces a state variable to store the input value f. This is because f remains in scope until the recursive call to *Idle*, and, as is the case here, may be referred to. Whenever an event with an input parameter is not immediately followed by a recursive call, then a new state variable will be introduced to the generated B machine to store that input parameter.

The actual value for a process index in a recursive call may be any B expression. Furthermore, IF – THEN – ELSE statements may be used in process descriptions and the guard may be any B predicate. This is illustrated by the following example:

```
MACHINE  $BUFFER(T)$
ALPHABET  $In(x : T)$      $x \longleftarrow Out$
    PROCESS  $InitBuffer \ = \ Buffer([\,])$  WHERE
        $Buffer(s : \mathrm{seq}\ T) \ =$
            IF  $s = [\,]$
            THEN  $In?x \rightarrow Buffer([x])$
            ELSE     $In?x \rightarrow Buffer(s \frown [x])$
                   $[]$  $Out!first(s) \rightarrow Buffer(tail(s))$
    END
END .
```

[3]While a guard represents an enabling condition, a precondition represents a termination condition.

Here, $[\,]$ represents the empty sequence, $[x]$ represents a singleton sequence, and $s \frown [x]$ represents the concatenation of s and $[x]$.

Input parameters may also be 'dot' parameters, which means that a process is only willing to accept a particular value as input rather than being willing to accept any value. This is illustrated by the following CSP example:

$$Free \;=\; Lock?u \rightarrow Locked(u)$$

$$\begin{aligned} Locked(v : USER) \;=\; & \quad Access.v \rightarrow Locked(v) \\ & \;[]\; Unlock.v \rightarrow Free\ . \end{aligned}$$

Here, any user u may lock some shared resource. Once it has been locked by u only that user may access or unlock the resource. A dot argument for an event may be any B expression and corresponds to an input parameter of an operation. A clause is added to the guard of the generated operation to constrain the input parameter so that it equals the dot value. For example, the *Access* operation generated from the above CSP would be:

$$Access(u) \;\hat{=}\; \text{SELECT}\;\; P = Locked \wedge u = v \;\; \text{THEN}\;\; skip \;\; \text{END}.$$

In the case that a CSP specification consists of more than one process, each output parameter of an operation may be determined by at most one of the processes, though different processes may determine different output parameters for the same operation.

2.4 Interleaving

A process may be defined as an interleaved composition of a set of indexed instances of a process. This is illustrated in the following example:

```
MACHINE  $MultiFileTransfer$
ALPHABET   $Send(u : User,\ v : User,\ f : File)$
                 $f \longleftarrow Receive(u : User,\ v : User)$
    PROCESS  $MultiCopy \;=\; |||\, u, v . Copy[u, v]$  WHERE
        $Copy[u, v] \;=\; Send.u.v?f \rightarrow Receive.u.v!f \rightarrow Copy[u, v]$
    END
END .
```

Here, *MultiCopy* is defined as the interleaved composition of an indexed set of instances of *Copy*, where *Copy* is indexed by a pair of variables u, v, both of type *User*. The indexing variables for the interleaving are placed in square brackets rather than round brackets because they are treated differently to standard process parameters. In the translation to B, the state of the process is represented by a function from the indexing set to the appropriate control type and an operation refers to the point in this function determined by its indexing input parameters. With the above example, two functions, representing the control states and the input parameter f, are generated as state variables. Both these functions take pairs of users as arguments corresponding to the indexing parameters of *Copy*. Each operation is indexed by a pair of users and accesses the functions at a point determined by the indexing pair of users. The generated machine is illustrated in Figure 2.

2.5 Conjunction

A CSP machine may be used to constrain the execution order of a standard B machine. Consider the B machine shown in Figure 3 which contains a variable representing a counter and operations for incrementing, decrementing and reading the counter.

The ordering of these operations may be further constrained by the CSP specification shown in Figure 4. This process description forces a user v to lock the counter before it can be manipulated by v and prevents other users from manipulating it while it is locked by v. Notice that the process

```
MACHINE  $MultiFileTransfer$
SETS  $MultiCopyState \; = \; \{ \; Copy, \; \; Copy\_1 \; \}$
VARIABLES  $MultiCopy, f\_1$
INVARIANT
    $MultiCopy \in (User \times User) \rightarrow MultiCopyState \;\; \wedge$
    $f\_1 \in (User \times User) \rightarrow File$
INITIALISATION  $MultiCopy := \lambda\, u, v.(u \in User \wedge v \in User \mid Copy)$

OPERATIONS

    $Send(u, v, f) \;\hat{=}$
        PRE  $u \in User \wedge v \in User \wedge f \in File$  THEN
            SELECT  $MultiCopy(u, v) = Copy$
            THEN  $MultiCopy(u, v) := Copy\_1 \quad || \quad f\_1(u, v) := f$
            END
        END;

    $f \longleftarrow Receive(u, v) \;\hat{=}$
        PRE  $u \in User \wedge v \in User$  THEN
            SELECT  $MultiCopy(u, v) = Copy\_1$
            THEN  $MultiCopy(u, v) := Copy \quad || \quad f := f\_1(u, v)$
            END
        END
END
```

Figure 2: Generated machine for *MultiFileTransfer*.

description places no constraint on the x parameter of the *Inc* and *Dec* operations nor does it constrain the *Read* operation.

The CONJOINS clause in Figure 4 signifies that the CSP specification is constraining the B machine *CounterActs*. For each event name *OpName* in the alphabet of the CSP specification, the conjoined machine should have a corresponding operation called *OpName_Act* as shown in the *CounterActs* machine of Figure 3.

The B machine generated by csp2B from the *Counter* specification of Figure 4 will include the *CounterActs* machine using the machine inclusion mechanism of B. Each operation, *OpName*, in the generated machine will include a guarded call to the corresponding *OpName_Act* operation of the included machine. That is, if S represents the composition of the statements generated from the various CSP processes for *OpName* and T represents the call to the corresponding operation of the conjoined machine, then *OpName* will have the form:

$$S \quad || \quad \text{SELECT} \;\; grd(S) \;\; \text{THEN} \;\; T \;\; \text{END} \; .$$

For example, the *Inc* operation in the B machine generated from *Counter* will be as follows:

```
$Inc(u, x) \;\; =$
    PRE  $u \in USER \;\wedge\; x \in \mathbb{N}$  THEN
        SELECT  $Locking = Locked \wedge u = v$  THEN  $skip$  END
      ||
        SELECT  $Locking = Locked \wedge u = v$  THEN  $Inc\_Act(x)$  END
    END;
```

The guarding of the call in the generated operation ensures that the composite statement is

```
MACHINE  $CounterActs(USER)$
VARIABLES  $c$
INVARIANT  $c \in \mathbb{N}$
INITIALISATION  $c := 0$

OPERATIONS

    $Lock\_Act(u) \;\hat{=}\;$ PRE  $u \in USER$  THEN  $skip$ END;

    $Inc\_Act(u,x) \;\hat{=}\;$ PRE  $u \in USER \wedge x \in \mathbb{N}$  THEN  $c := c + x$ END;

    $Dec\_Act(u,x) \;\hat{=}$
        PRE  $u \in USER \wedge x \in \mathbb{N}$  THEN
            SELECT  $c \geq x$  THEN  $c := c - x$ END
        END;

    $Unlock\_Act(u) \;\hat{=}\;$ PRE  $u \in USER$  THEN  $skip$ END;

    $y \longleftarrow Read\_Act(u) \;\hat{=}\;$ PRE  $u \in USER$  THEN  $y := c$ END

END
```

Figure 3: B part of counter specification.

enabled exactly when both S and T are enabled since, provided $trm(S) = true$:

$$grd(\ S \ \ || \ \text{SELECT} \ \ grd(S) \ \ \text{THEN} \ \ T \ \ \text{END} \ ) \quad = \quad grd(S) \wedge grd(T) \ .$$

S is generated by csp2B and it will always be the case that $trm(S) = true$[4].

2.6 Variable Access

The variables of the conjoined machine may be referred to (read only) in any expressions of the CSP specification. For example, consider the *BUFFER_Acts* machine of Figure 5. This machine is conjoined with the *BUFFER* CSP specification of Figure 6 which means that its variable s may be referred to in the CSP process. In the CSP specification, s is referenced in the guard of the IF-statement defining the behaviour of *Buffer* and in the expression determining the output value for the *Out* channel.

2.7 Refinements

A CSP description may be a refinement of another (CSP or B) machine. As in B, the keyword REFINEMENT is used instead of MACHINE and the REFINES clause must identify the machine being refined.

The abstraction invariant for the refinement may be placed in the INVARIANT clause of the CSP machine. Before devising the abstraction invariant, it is usually convenient to generate the B machine from the CSP machine. This will make explicit the control states and state transitions generated from the CSP description. These may then be used in the abstraction invariant for the refinement.

[4] Strictly speaking, a statement constructed by csp2B will terminate only when its input parameters are correctly typed.

```
MACHINE $Counter$
CONJOINS $CounterActs(USER)$
SETS $USER$
ALPHABET   $Lock(u : USER)$   $Unlock(u : USER)$
           $Inc(u : USER,\ x : \mathbb{N})$   $Dec(u : USER,\ x : \mathbb{N})$
           $y \longleftarrow Read(u : USER)$

    PROCESS $Locking = Free$
    CONSTRAINS $Lock(u)$ $Inc(u)$ $Dec(u)$ $Unlock(u)$ WHERE
        $Free = Lock?u \rightarrow Locked(u)$
        $Locked(v : USER) =$
              $Inc.v \rightarrow Locked(v)$
            $\Box$ $Dec.v \rightarrow Locked(v)$
            $\Box$ $Unlock.v \rightarrow Free$
    END
END
```

Figure 4: CSP part of counter specification.

```
MACHINE $BUFFER\_Acts(T)$
VARIABLES $s$
INVARIANT $s \in \text{seq } T$
INITIALISATION $s := [\,]$

OPERATIONS
    $In\_Act(x)$ $\hat{=}$ PRE $x \in T$ THEN $s := s \frown [x]$ END;

    $Out\_Act$ $\hat{=}$ PRE $s \neq [\,]$ THEN $s := tail(s)$ END
END
```

Figure 5: B part of buffer.

Figures 7 and 8 give an example of a refinement of this form. The CSP part is described in Figure 7 and this specification is conjoined with the B machine of Figure 8. This is a refinement of the *FileTransfer* machine of Figure 1 and it transfers a file byte-by-byte instead of in one step. In the refinement, an incoming file is stored in the variable *afile* while an outgoing file comes from the variable *bfile*. The contents of *afile* are gradually transferred to *bfile* using a byte-wise transmission protocol. The CSP part describes the ordering of events whereby both sides in a transmission agree to start a transmission (*OpenReq* followed by *OpenResp*), then transfer each bit using a succession of *Trans* events and then finish the transmission. Operations such as *OpenReq* that do not appear in the abstract specification *FileTransfer* nor in the conjoined machine *FileTransferRefinement* are all *skip* actions and have been omitted here.

The refinement invariant used to prove the refinement is included in the CSP specification of Figure 7. This invariant refers to the abstract and concrete control variables (*Copy* and *ByteWise* respectively) and, when deriving the invariant, it was useful to be able to perform the translation of the CSP description in order to see an explicit representation of its state and state transitions. The invariant is then added to the CSP specification and the translation performed again. The invariant is copied over into the generated B machine and is used by the proof obligation generator of a B tool.

```
MACHINE  $BUFFER(T)$
CONJOINS  $BUFFER\_Acts(T)$
ALPHABET  $In(x \in T)$     $x \longleftarrow Out$

     PROCESS  $InitBuffer \; = \; Buffer$
     CONSTRAINS  $In$     $y \longleftarrow Out$  WHERE
          $Buffer \; =$
               IF  $s = [\,]$
               THEN  $In \rightarrow Buffer$
               ELSE  $In \rightarrow Buffer$    $[]$    $Out!first(s) \rightarrow Buffer$
     END
END
```

Figure 6: CSP part of buffer.

3 Semantic Issues

In [9], Hoare takes a denotational approach to the semantics of CSP by defining the Failure-Divergences model for processes. It is also possible to take an operational approach by considering processes as Labelled Transition Systems (LTS) in the manner of Milner's CCS [10]. The B machine generated from a CSP specification may be viewed as an LTS and, for this reason we, take an operational approach to justifying the semantics of the csp2B translation. Roscoe [12] shows how the denotational and operational models of CSP are linked.

In the absence of CSP processes being allowed to access the state variables of conjoined B machines, the semantics is entirely compositional. That is, the semantics of the combination of several CSP processes conjoined with a B machine (viewed as an LTS) is precisely CSP parallel composition. This entails monotonicity of refinement allowing the B machine to be refined independently. However, as we shall see, this compositionality fails when sharing of state variables occurs. First we consider the case where no sharing of variables occurs and look at normal forms for CSP processes and how they define an LTS.

3.1 Normal Form

In the notation supported by csp2B, a process is described by a set of equations of the form

$$I_i(v) \;=\; P_i \;,$$

where I_i is a process identifier and P_i is a process term which may contain several process identifiers. A process term P is said to be in *normal form* either if $P = STOP$ or if P is a choice in which each branch is a boolean-guarded event prefixing a recursive call to another process identifier, that is,

$$P \;=\; Q_1 \;\; [] \;\; \cdots \;\; [] \;\; Q_n \;,$$

where each Q_i is of the form

$$\text{IF} \;\; G \;\; \text{THEN} \;\; a \rightarrow I(e) \;.$$

Here I must be the identifier on the left of a process equation and not a more complicated process term. The event a may contain input, output and dot parameters. We assume that choice is associative and commutative.

There is an important syntactic restriction in csp2B which requires that all recursive calls are prefixed by an event and this includes recursive calls in the branches of an IF-statement. This ensures that any set of syntactically-correct process equations may be transformed to normal form.

```
MACHINE  $\mathit{FileTransferRefinement}$
REFINES  $\mathit{FileTransfer}$
CONJOINS  $\mathit{FileTransferRefinementActs}(\mathit{Byte})$

INVARIANT
    $(\mathit{ByteWise} \in \{BW_1, BW_2\} \;\Rightarrow\; g = \mathit{afile}) \;\wedge$
    $(\mathit{ByteWise} = \mathit{Transfer} \;\Rightarrow\; g = \mathit{bfile} \frown \mathit{afile}) \;\wedge$
    $(\mathit{ByteWise} = \mathit{Transfer\_1} \;\Rightarrow\; g = \mathit{bfile})$
    $\wedge$
    $(\mathit{Copy} = \mathit{Idle} \;\Leftrightarrow\; \mathit{ByteWise} \in \{BW, \mathit{Transfer\_2}\}) \;\wedge$
    $(\mathit{Copy} = \mathit{Remember} \;\Leftrightarrow\; \mathit{ByteWise} \in \{BW\_1, BW\_2, \mathit{Transfer}, \mathit{Transfer\_1}\})$

ALPHABET   $\mathit{Send}(f : \mathit{File})$    $f \longleftarrow \mathit{Receive}$
           $\mathit{OpenReq}$   $\mathit{OpenResp}$   $\mathit{TransBlock}$   $\mathit{EndTrans}$   $\mathit{Ack}$

    PROCESS  $\mathit{ByteWise} \;=\; BW$  WHERE
        $BW \;=\; \mathit{Send} \rightarrow \mathit{OpenReq} \rightarrow \mathit{OpenResp} \rightarrow \mathit{Transfer}$
        $\mathit{Transfer} \;=$
            IF  $\mathit{afile} = [\;]$
            THEN  $\mathit{EndTrans} \rightarrow \mathit{Receive} \rightarrow \mathit{Ack} \rightarrow BW$
            ELSE  $\mathit{TransBlock} \rightarrow \mathit{Transfer}$
    END
END
```

Figure 7: CSP part of *FileTransfer* refinement.

IF-statements are distributed through a term using the following transformations:

$$\begin{aligned}
\text{IF } G \text{ THEN } P \text{ ELSE } Q \quad &= \quad (\text{IF } G \text{ THEN } P) \;\square\; (\text{IF } \neg\, G \text{ THEN } Q) \\
\text{IF } G \text{ THEN } (P \;\square\; Q) \quad &= \quad (\text{IF } G \text{ THEN } P) \;\square\; (\text{IF } G \text{ THEN } Q) \\
\text{IF } G \text{ THEN } (\text{IF } H \text{ THEN } P) \quad &= \quad (\text{IF } G \wedge H \text{ THEN } P)\;.
\end{aligned}$$

Nested prefixing is normalised by introducing new equations. An equation of the form

$$I(v) \quad = \quad (\text{IF } G \text{ THEN } a \rightarrow P) \;\square\; Q\;,$$

where P is not a process identifier, is replaced by the pair of equations

$$\begin{aligned}
I(v) \quad &= \quad (\text{IF } G \text{ THEN } a \rightarrow J(v, w)) \;\square\; Q \\
J(v, w) \quad &= \quad P
\end{aligned}$$

Here, J is some fresh process identifier and w is the list of input parameters in the event a (with renaming to fresh variables where necessary to avoid name clashes). The introduction of w is necessary because P may refer to any input parameters of a.

Terms involving $STOP$ are simplified as follows:

$$\begin{aligned}
\text{IF } G \text{ THEN } STOP \quad &= \quad STOP \\
STOP \;\square\; P \quad &= \quad P.
\end{aligned}$$

In the case that a set of process equations has different parameter lists, then the parameter lists are extended to the merge of all the parameters. For example, a pair of process equations

$$\begin{aligned}
I(v) \quad &= \quad a \rightarrow J(e) \\
J(w) \quad &= \quad b \rightarrow I(f)\;,
\end{aligned}$$

```
MACHINE  $FileTransferRefinementActs(Byte)$
VARIABLES  $afile, bfile$
INVARIANT  $afile \in File \;\wedge\; bfile \in File$

OPERATIONS

    $Send\_Act(f) \;\hat{=}\;$ PRE $f \in File$ THEN $afile := f$ END;

    $f \longleftarrow Receive\_Act \;\hat{=}\; f := bfile$;

    $OpenResp\_Act \;\hat{=}\; bfile := [\,]$;

    $TransBlock\_Act \;\hat{=}\; bfile := bfile \frown (first\;\, afile) \;\;||\;\; afile := tail(afile)$

END
```

Figure 8: B part of *FileTransfer* refinement.

becomes

$$\begin{aligned} I(v, w) &= a \to J(v, e) \\ J(v, w) &= b \to I(f, w)\ . \end{aligned}$$

3.2 Labelled Transition Systems

A process specification consisting of a set of normal-form equations defines an LTS. The state space is the cartesian product of the set of process identifiers with the type of the indexing parameters. We continue to write elements of this state space as $I(v)$. The labels of the LTS are the parameterised event names. A label may consist of several components $c.i.j.k$ and an event of the form $c.i?y!k$ stands for the set of labels $\{\ c.i.j.k \ \mid\ j \in Y\ \}$, where Y is the type of input parameter y. The LTS may make a transition labelled $a.i.j.k$ from state $I(v)$ to $I'(v')$, written

$$I(v) \xrightarrow{a.i.j.k} I'(v')\ , \tag{1}$$

if the set of normalised equation contains an equation of the form

$$I(v) \;=\; \cdots \;\;[]\;\; \text{IF}\;\; G(v)\;\; \text{THEN}\;\; a.i?y!k \to I'(V) \;\;[]\;\; \cdots\ , \tag{2}$$

and $G(v)$ holds and $v' = V[y := j]$.

In converting a CSP specification to B, csp2B normalises the set of equations as described previously and then constructs a B machine corresponding to the LTS. The state space is represented by the state variables of the machine, the labels are represented by the operation names (along with input and output parameters) and the transitions are represented by the operations. The B machine contains state variables (v) corresponding to the list of indexing parameters as well as a special control variable (p) typed over the set of process identifiers from the left hand sides of the equations. For each event name a in the alphabet of the CSP process, the B machine contains an operation of the form

$$z \longleftarrow a(x, y) \;\hat{=}\; S_a\ ,$$

where S_a is constructed in the following manner: For each occurrence of event a in a normalised CSP process equation of the form (2), the B operation has a SELECT branch of the form:

$$\text{SELECT}\;\; G(v) \wedge p = I \wedge x = i\;\; \text{THEN}\;\; p, v, z := I', V, k\;\; \text{END}\ . \tag{3}$$

The clause $x = i$ ensures that the input value x matches the dot value i. All the branches of S_a are composed using the B choice operator $S \, [] \, T$.

We briefly outline why the LTS defined by the set of normalised equations is the same as the LTS defined by the constructed B machine. In order to define when a transition is allowed by a B operation, we use the notion of conjugate weakest precondition defined as follows [11]:

$$\langle S \rangle Q \mathrel{\hat{=}} \neg \, [S] \neg \, Q \; .$$

$\langle S \rangle Q$ represents the weakest precondition under which it is possible for S to establish Q (as opposed to the guarantee provided by $[S]Q$). If the machine contains an operation of the form

$$z \longleftarrow a(x, y) \mathrel{\hat{=}} S_a$$

then the transition

$$I(w) \xrightarrow{a.i.j.k} I'(w') \tag{4}$$

is possible in state $I(w)$ provided

$$\langle \; p, v, x, y := I, w, i, j \; ; \;\; S_a \; \rangle \; (p, v, z = I', w', k) \tag{5}$$

holds. That is, it is possible for S_a to establish an outcome in which p and v equal $I'(w')$ and z equals k when p, v, x, y are initialised appropriately.

Now, a process equation of the form (2) enables a transition of the form (4) in state $I(w)$ when $G(w)$ holds. Using (5), it is easy to show that this is precisely the same condition under which the choice branch (3) allows this transition. Furthermore transition (4) is allowed if there is some occurrence of event a in some normalised CSP process equation. Likewise, the constructed B operation allows the transition if there is some choice branch that allows it which follows from:

$$\langle S \, [] \, T \rangle Q \; = \; \langle S \rangle Q \; \vee \; \langle T \rangle Q \; . \tag{6}$$

Thus the transition relation $\xrightarrow{a.i.j.k}$ defined by the set of normalised CSP process equations is the same as the transition relation defined by the constructed B machine.

3.3 Interleaving

The csp2B tool supports interleaving of processes at the outermost level only. This interleaving has the form $||| \; i.P[i]$ where all of the instances $P[i]$ behave in a similar way except for the indexing of event labels by i. Such an interleaving represents multiple instance of $P[i]$ running in parallel where the parallel instances do not interact with each other in any way. This interleaving is modelled as a single large LTS whose state is modelled by replicated instances of the state that a single $P[i]$ would normally have. Thus, if the LTS for a single $P[i]$ would normally have a state space Σ and i ranges over an indexing set I, then the state space of the large LTS is $I \to \Sigma$. This is the basis for the translation to B of interleaving described in Section 2.4.

3.4 Compositionality

The parallel composition of two LTS's P and Q is an LTS $P \parallel Q$ formed by taking the cartesian product of their state spaces and merging common actions. If a is common to P and Q, then their composition has a transition labelled a as defined by the following rule:

$$\frac{I \xrightarrow{a} I' \; \in \; P \quad \wedge \quad J \xrightarrow{a} J' \; \in \; Q}{(I, J) \xrightarrow{a} (I', J') \; \in \; P \parallel Q \; .} \tag{7}$$

This models synchronised parallelism since both P and Q must be in states that enable a for a to be enabled in $P \parallel Q$. Events that are present in only one of the processes result in transitions that have no effect on the other process. Thus, if a is an event of P but not of Q, then the composition has a transition defined by the following rule:

$$\frac{I \xrightarrow{a} I' \in P}{(I, J) \xrightarrow{a} (I', J) \in P \mid\mid Q\ .}$$

Similarly for the case where a is an event of Q only.

When generating a B machine from parallel processes and a conjoined machine, csp2B composes the appropriate statements using the B parallel operator. Thus, given two parallel processes P and Q, csp2B constructs an operation of the form $S \mid\mid T$ for the generated B machine, where S is constructed from P and T is constructed from Q in the usual way. The following result about the B parallel operator is important in showing that this corresponds to the LTS definition of parallel composition: Let S be a statement that assigns to x only and let T be a statement that assigns to y only. Let M be a predicate that depends on x only and let N be a statement that depends on y only. If $trm(S) = trm(T) = true$, then

$$\langle S \mid\mid T\rangle(M \wedge N) \quad = \quad \langle S\rangle M \ \wedge\ \langle T\rangle N\ . \tag{8}$$

Using this result, it is easy to show that the relationship between transitions allowed by S and T and those allowed by $S \mid\mid T$ is precisely that of (7). Thus, in the absence of variable sharing, the parallel composition used by csp2B corresponds to the CSP definition of parallelism.

An important consequence of this result is that, since CSP parallel composition is monotonic with respect to refinement, a conjoined machine may be refined separately while maintaining the refinement of the overall system. Refinement of CSP processes is defined in terms of the Failures-Divergences model. Based on [11, 13], [5] defines the Failures-Divergences semantics of B machines and shows that refinement of B machines corresponds to refinement at the Failures-Divergences level[5].

3.5 Divergence and Nontermination

In the above presentation, we have assumed that systems never diverge. This will always be the case for CSP processes written in the notation supported by csp2B since it does not contain a hiding operator. However, Morgan [11] shows that it is appropriate to equate nontermination of operations with divergent behaviour and the operations of a conjoined B machine may be nonterminating in some states, e.g., operations of the form PRE M THEN S END. This situation may be dealt with by introducing a special bottom state modelling divergence to the LTS model along with several extra transition rules. Alternatively, one may directly define the failures and divergences of a B machine using conjugate weakest-precondition formulae in the manner of [11]. (We have found this latter approach to be the most convenient.) Either way one can show that the correspondence between CSP parallelism, and the composition used in the construction of B machines by csp2B, holds. Such a proof is presented in detail in [4]. We presented an LTS-style justification in this paper since it is simpler (though less comprehensive) and portrays the essence of the translation.

The potential presence of nontermination in a conjoined operation makes it essential to guard calls to the conjoined machine, i.e., if S is the operation constructed from a CSP process and T is a call to the conjoined machine, then the operation in the generated B machine has the form:

$$S \mid\mid \text{SELECT}\ \ grd(S)\ \ \text{THEN}\ \ T\ \ \text{END}\ . \tag{9}$$

If T was not guarded by $grd(S)$, then the composition of S and T would be enabled in any state in which T is nonterminating, even if S is not enabled in that state. This arises from the definition of $S \mid\mid T$, and we have, for example, that

$$\text{SELECT}\ \ false\ \ \text{THEN}\ \ skip\ \ \text{END}\ \ \mid\mid\ \ abort \quad = \quad abort\ .$$

[5] Strictly speaking, for this correspondence to hold, an extra condition on refinement of B machines is introduced which requires that the guard of an abstract operation implies the guard of a concrete operation.

Guarding T avoids the possibility of the composition being enabled in states where S is not enabled. It is not necessary to guard S since it is constructed from a nondivergent CSP process and will therefore be fully terminating.

3.6 Variable Sharing

In the case where a CSP process refers to the state variables of a conjoined machine, the compositionality result no longer holds. This is because the CSP processes cannot be given an independent CSP semantics if they refer to variables outside their control. Interaction between processes in CSP is based purely on interaction via synchronised events and allowing them to access the state of another machine would allow for stronger interaction that just synchronisation over shared events.

In the case where sharing of variables occurs, the semantics of the whole system is given by normalising the CSP processes in the usual way and collapsing the results, along with the conjoined machine, into a single large LTS. This single large then needs to be refined as a whole. Of course the modularity provided by B for structuring developments can still be availed of, and the whole system does not have to be represented as a single B machine, rather its semantics are those of a single LTS.

4 Conclusions

We have presented an outline of the functionality of the csp2B tool and provided an operational-semantic justification for the way in which it translates CSP to B. The supported CSP notation provides a powerful way of describing ordering constraints for reactive system development and enhances the standard B notation. The tool provides a useful extension to the B Method and can easily be used in conjunction with existing B tools.

An interesting feature of the tool is that it accepts expressions, types and predicates written in standard B notation, copying them directly to the generated B machine. This means that it supports quite a rich CSP notation.

There are features of CSP that are not supported by the tool, namely internal nondeterministic choice, event hiding and arbitrary (i.e., not just at the outermost level) parallel composition and interleaving. Supporting these features would result in a lot of complexity in the generated B machines. For example, extra flag variables would have to be introduced to model the CSP internal choice operator. Some of these features can be achieved directly in the B part of a specification. Internal choice can be modelled in the B part using nondeterministic constructs of B. Alternatively, one could take a CCS-like approach and represent the internal choice of P and Q as the process

$$(i \rightarrow P) \ [\,] \ (i \rightarrow Q),$$

where i is regarded as a hidden event. Event hiding may be modelled using the notion of internal operations in B machines as introduced in [5]. The tool has been applied to a larger example (a form of distributed database) [8] than those presented here and the restrictions in the supported CSP notation did not prove a hindrance.

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] R.J.R. Back and R. Kurki-Suonio. Decentralisation of process nets with centralised control. In *2nd ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, pages 131–142, 1983.

[4] M.J. Butler. *A CSP Approach To Action Systems.* D.Phil. Thesis, Programming Research Group, Oxford University, 1992.

[5] M.J. Butler. An approach to the design of distributed systems with B AMN. In J.P. Bowen, M.G. Hinchey, and D. Till, editors, *10th International Conference of Z Users (ZUM'97)*, volume LNCS 1212, pages 223 – 241. Springer–Verlag, April 1997.

[6] M.J. Butler. csp2B User Manual. Available from `http://www.ecs.soton.ac.uk/~mjb/csp2B`, 1999.

[7] M.J. Butler and M. Waldén. Distributed system development in B. In H. Habrias, editor, *First B Conference*, November 1996.

[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. `http://www.dsse.ecs.soton.ac.uk/techreports/99-1.html`.

[9] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice–Hall, 1985.

[10] R. Milner. *Communication and Concurrency.* Prentice–Hall, 1989.

[11] C.C. Morgan. 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–Verlag, 1990.

[12] A.W. Roscoe. *The Theory and Practice of Concurrency.* Prentice–Hall, 1998.

[13] J.C.P. Woodcock and C.C. Morgan. Refinement of state-based concurrent systems. In D. Bjørner, C.A.R. Hoare, and H. Langmaack, editors, *VDM '90*, volume LNCS 428, pages 340–351. Springer–Verlag, 1990.