

# csp2B: A Practical Approach To Combining CSP and B

Michael Butler

Department of Electronics & Computer Science  
University of Southampton  
United Kingdom.

**Keywords:** CSP; B Method; Refinement

## **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, 6, 8]. 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. Events are guarded by conditions on the state and may only be executed when their guard is enabled.

---

*Correspondence and offprint requests to:* Michael Butler, Department of Electronics & Computer Science, University of Southampton, Highfield, Southampton SO17 1BJ, United Kingdom. [M.J.Butler@ecs.soton.ac.uk](mailto:M.J.Butler@ecs.soton.ac.uk).

Action systems may be refined using standard techniques for program refinement and data refinement. A system  $N$  is said to be a refinement of system  $M$  if any behaviour exhibited by  $N$  is a possible behaviour of  $M$ . Action system refinement techniques includes the possibility of introducing auxiliary, or internal, actions in a refinement step. This feature is useful in the design of distributed systems as it allows one to start with a specification that is independent of the distributed architecture, in which all information is globally available, and then use data refinement to partition the information, introducing internal actions to transfer information between partitions. An example of this may be found in [5], where a mail service is specified using a single mail bag to which users add addressed messages when sending mail and from which they may read mail; this simple ‘global view’ is then refined into a system with a more complicated data structure representing a network of nodes while internal actions are introduced to model the routing of messages through the network.

The ability to decompose action systems into parallel subsystems is also important for the design of distributed systems. This can be done in two ways: partition the actions amongst the subsystems and allow the subsystems to interact using shared state [4], or partition the state and allow interaction through shared actions [5]. Action system refinement and decomposition provide a powerful and practical approach to the correct design of distributed systems and they may be used to enhance industrial-strength state-based methods such as Z and B (see, for example, [6, 8]). In particular, parallel composition/decomposition is essentially a syntactic procedure.

However, while B machines (and action systems) are good at modelling parallel activity, they are 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 [12] and CCS [14]. The designers of the languages  $SL_0$  [16] and CSP-OZ [10] have gone some way towards addressing this by combining CSP-like process specifications with action system-like specifications. However, while they allow for the separate refinement of the process parts and the action system parts, they do not support refinement steps involving the introduction or elimination of process parts nor allow one to make use of the process part when refining the action system part.

This ability to mix the process and the action system parts in refinements, as well as specification, is the contribution of the csp2B tool. 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.

### 1.1. B Abstract Machine Notation (AMN)

B AMN is a model-oriented formal notation and is part of the B-method developed by Abrial [1]. A system in B is specified as an abstract machine which has the form outlined in Fig. 1. An abstract machine consists of some sets ( $SS$ ), constants ( $C$ ) and variables ( $v$ ) which are modelled using standard set-theoretic constructs. Properties ( $P$ ) and invariants ( $I$ ) are first-order predicates. Operations act on the variables while preserving the invariant and have input parameters ( $x_i$ ) and output parameters ( $y_i$ ).

MACHINE $M$
SETS $SS$
CONSTANTS $C$
PROPERTIES $P$
VARIABLES $v$
INVARIANT $I$
INITIALISATION $init$
OPERATIONS
$y_1 \leftarrow op_1(x_1) \hat{=} S_1$
$y_2 \leftarrow op_2(x_2) \hat{=} S_2$
...
END

**Fig. 1.** Abstract machine outline.

In the AMN operation notation, changes to the state variables are specified using assignment statements ( $v := E$ ). A simultaneous assignment of the form  $x := E \parallel y := F$  is equivalent to  $x, y := E, F$ .

A guarded action is written as a statement of the form:

SELECT  $G$  THEN  $S$  END,

where the guard  $G$  is some condition on the state variables and  $S$  is some statement. Such an action is enabled only when  $G$  holds. This provides a means of specifying reactive systems in which operations are only enabled in certain states. The nondeterministic choice of two statements is written  $S \square T$ . Such a statement is enabled when either  $S$  or  $T$  are enabled.

We write  $grd(S)$  for the enabling condition of a statement  $S$ . The rules shown in Fig. 2 may be used to calculate the enabling condition of a compound statement. As well as an enabling condition, statements in B have a separate termination condition, written  $trm(S)$ , representing the condition under which execution of a statement is guaranteed to terminate. In this paper and in the csp2B tool, we use a subset of the B statement language in which statements always terminate, i.e.,  $trm(S) = true$ . This means  $grd(S \parallel T)$  may be simplified to  $grd(S) \wedge grd(T)$ . Normally the B tools use termination conditions to give types to input parameters of operations so that, strictly speaking, operations will only terminate if input parameters are correctly typed. For example, in B input parameter  $x$  is given type  $T$  using a preconditioned statement as follows:

$op(x) \hat{=} \text{PRE } x \in T \text{ THEN } \dots \text{ END.}$

The precondition  $x \in T$  is regarded as a termination condition in B. However, to ensure that statements are always terminating, we simply regard  $x \in T$  as type information that is used to define the semantics of machines as labelled transition systems (see Section 4.2).

## 1.2. CSP

In CSP, systems are modelled as processes. Associated with each process is an alphabet of communication events representing the visible interface between the process and its environment. Each process has a behaviour specification which

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

**Fig. 2.** Calculating statement guards.

constrains the way in which the process can engage in the events of its alphabet. The behaviour of a process is specified using the CSP process term language.

The term *STOP* represents the process that refuses to engage in any event. Sequencing of events is described by the prefix operator ( $\rightarrow$ ): the term  $a \rightarrow P$  describes the process that engages in the event  $a$  and then behaves as process  $P$ . External choice of behaviour is described by the choice operator ( $\parallel$ ):  $P \parallel Q$  represents the process that offers the choice to the environment between behaving as  $P$  or as  $Q$ . CSP also has a separate internal-choice operator that causes a process to internally choose between behaving as one way or another, leaving the environment with no control over that choice. The CSP hiding operator internalises events, leaving the environment with no control over those events.

The parallel composition of two CSP process  $P$  and  $Q$  is written  $P \parallel Q$ .  $P$  and  $Q$  interact by synchronising over shared events, that is, an event common to  $P$  and  $Q$  can occur in  $P \parallel Q$  only in situations in which both  $P$  and  $Q$  permit that common event. With interleaving of processes, written  $P \parallel\parallel Q$ , events common to  $P$  and  $Q$  occur independently so that there is no synchronisation.

The csp2B tool supports a CSP process notation containing prefixing ( $\rightarrow$ ), choice ( $\parallel$ ) and the deadlocked process *STOP*. It does not support an internal choice 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, Section 3 gives an overview of refinement in B and how it is supported by csp2B, and Section 4 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<sup>1</sup>):

```
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 [12]). 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 /* Generated by csp2B */
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 .
```

The generated B machine contains a ‘control’ variable named *VM*, the same as

---

<sup>1</sup> The tool supports an ascii version of CSP and the full syntax may be found in [7].

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.

## 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:

$$AwaitCoin = Coin \rightarrow (Tea \rightarrow AwaitCoin \parallel Coffee \rightarrow AwaitCoin) .$$

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:

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

The generated  $Coin$  operation, for example, will then be:

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

## 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
  CONSTRAINTS Tea Coffee WHERE
    Alt = Coffee → Tea → Alt
  END
END .
```

In CSP terms, *VendingMachine* represents the parallel composition of *VM* and *Alternate*. The (optional) *CONSTRAINTS* 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:

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

Recall that  $grd(S \parallel T) = grd(S) \wedge grd(T)$ . 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 4.4).

### 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:

$$\begin{aligned} Idle &= In?f \rightarrow Remember(f) \\ Remember(f) &= Out!f \rightarrow Idle . \end{aligned}$$

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:

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

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 .
```

The indexing variable  $g$  in this specification becomes a state variable of type  $File$  in the generated B machine shown in Fig. 3. 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.

```

MACHINE FileTransfer /* Generated by csp2B */
SETS Byte; CopyState = { Idle, Remember }
VARIABLES Copy, g
DEFINITIONS File == seq(Byte)
INVARIANT Copy ∈ CopyState ∧ g ∈ File
INITIALISATION Copy := Idle
OPERATIONS
  Send(f) =
    PRE f ∈ File THEN
      SELECT Copy = Idle
      THEN Copy := Remember || g := f END
    END;
  f ← Receive =
    SELECT Copy = Remember
    THEN Copy := Idle || f := g END
END

```

**Fig. 3.** Generated machine for *FileTransfer*.

*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 ← Out
PROCESS InitBuffer = Buffer([ ]) WHERE
  Buffer(s : seq T) =
    IF s = [ ]
    THEN In?x → Buffer([x])
    ELSE ( In?x → Buffer(s ^ [x])
          || Out!first(s) → Buffer(tail(s)) )
END
END .

```

Here, [ ] represents the empty sequence, [*x*] represents a singleton sequence, and *s* ^ [*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:

$$\begin{aligned} \text{Free} &= \text{Lock}?u \rightarrow \text{Locked}(u) \\ \text{Locked}(v : \text{USER}) &= \text{Access}.v \rightarrow \text{Locked}(v) \\ &\quad \parallel \text{Unlock}.v \rightarrow \text{Free} . \end{aligned}$$

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:

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

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( $u : \text{User}, v : \text{User}, f : \text{File}$ )
            $f \leftarrow \text{Receive}(u : \text{User}, v : \text{User})$ 
PROCESS  MultiCopy =  $\parallel u, v. \text{Copy}[u, v]$  WHERE
            $\text{Copy}[u, v] = \text{Send}.u.v?f \rightarrow \text{Receive}.u.v!f \rightarrow \text{Copy}[u, v]$ 
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 Fig. 4.

## 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 Fig. 5 which contains a variable rep-

```

MACHINE MultiFileTransfer /* Generated by csp2B */
SETS MultiCopyState = { Copy, Copy_1 }
VARIABLES MultiCopy, f_1
INVARIANT
  MultiCopy ∈ (User × User) → MultiCopyState ∧
  f_1 ∈ (User × User) → File
INITIALISATION MultiCopy := λ u, v. (u ∈ User ∧ v ∈ User | Copy)

OPERATIONS

  Send(u, v, f) ≐
    PRE u ∈ User ∧ v ∈ User ∧ f ∈ File THEN
      SELECT MultiCopy(u, v) = Copy
      THEN MultiCopy(u, v) := Copy_1 || f_1(u, v) := f
      END
    END;

  f ← Receive(u, v) ≐
    PRE u ∈ User ∧ v ∈ User THEN
      SELECT MultiCopy(u, v) = Copy_1
      THEN MultiCopy(u, v) := Copy || f := f_1(u, v)
      END
    END

END

```

**Fig. 4.** Generated machine for *MultiFileTransfer*.

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 Fig. 6. 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 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 Fig. 6 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 Fig. 5.

The B machine generated by csp2B from the *Counter* specification of Fig. 6 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:

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

```

MACHINE CounterActs(USER)
VARIABLES c
INVARIANT c ∈ ℕ
INITIALISATION c := 0

OPERATIONS

  Lock_Act(u) ≐ PRE u ∈ USER THEN skip END;

  Inc_Act(u, x) ≐ PRE u ∈ USER ∧ x ∈ ℕ THEN c := c + x END;

  Dec_Act(u, x) ≐
    PRE u ∈ USER ∧ x ∈ ℕ THEN
      SELECT c ≥ x THEN c := c - x END
    END;

  Unlock_Act(u) ≐ PRE u ∈ USER THEN skip END;

  y ← Read_Act(u) ≐ PRE u ∈ USER THEN y := c END

END

```

Fig. 5. B part of counter specification.

```

MACHINE Counter
CONJOINS CounterActs(USER)
SETS USER
ALPHABET Lock(u : USER)  Unlock(u : USER)
          Inc(u : USER, x : ℕ)  Dec(u : USER, x : ℕ)
          y ← Read(u : USER)

PROCESS Locking = Free
CONSTRAINS Lock(u) Inc(u) Dec(u) Unlock(u) WHERE
  Free = Lock?u → Locked(u)
  Locked(v : USER) =
    Inc.v → Locked(v)
    □ Dec.v → Locked(v)
    □ Unlock.v → Free

END
END

```

Fig. 6. 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  $s := s \hat{\ } [x]$  END;

  Out_Act  $\hat{=}$  PRE  $s \neq []$  THEN  $s := \text{tail}(s)$  END
END

```

**Fig. 7.** B part of buffer.

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

```

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

```

The guarding of the call in the generated operation ensures that the composite statement is enabled exactly when both *S* and *T* are enabled since, provided  $\text{trm}(S) = \text{true}$ :

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

*S* is generated by csp2B and it will always be the case that  $\text{trm}(S) = \text{true}$ .

## 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 Fig. 7. This machine is conjoined with the *BUFFER* CSP specification of Fig. 8 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.

## 3. Refinement

In B, an abstract machine is refined by applying the standard technique of data refinement to its state: an abstraction invariant is used to relate the state variables of the abstract system to those of the refined system and data refinement should hold between correspondingly-named actions in the abstract and refined systems.

The refinement rules for B are expressed in terms of weakest-precondition

```

MACHINE BUFFER(T)
CONJOINS BUFFER_Acts(T)
ALPHABET In(x ∈ T)   x ← Out

PROCESS InitBuffer = Buffer
CONSTRAINS In  y ← Out WHERE
  Buffer =
    IF s = []
    THEN In → Buffer
    ELSE In → Buffer  []  Out!first(s) → Buffer
END
END

```

Fig. 8. CSP part of buffer.

$$\begin{aligned}
[skip]P &\hat{=} P \\
[SELECT\ G\ THEN\ S\ END]P &\hat{=} G \Rightarrow [S]P \\
[S\ []\ T]P &\hat{=} [S]P \wedge [T]P
\end{aligned}$$

Fig. 9. AMN weakest-precondition rules.

formulae: for action  $S$  and postcondition  $P$ , the formula  $[S]P$  (weakest precondition of  $S$  with respect to  $P$ <sup>2</sup>) characterises those initial states from which  $S$  is guaranteed to terminate in a state satisfying  $P$ . The semantics of assignment are simply substitution:  $[x := E]P$  stands for  $P$  with all free occurrences of  $x$  replaced by  $E$ . The semantics of several other AMN constructs are specified in Fig. 9 (note that this is only a subset of the full language).

$\langle S \rangle Q$  is the *conjugate* of  $[S]$ , representing the weakest precondition under which it is possible for  $S$  to establish  $Q$  (as opposed to the guarantee provided by  $[S]Q$ ) [15]. The conjugate weakest precondition defined as follows:

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

If  $S$  is a statement that acts on variables  $a$ ,  $T$  is a statement that acts on variables  $c$ , and  $AI$  is an abstraction invariant then we write

$$S \sqsubseteq_{AI} T$$

for ‘ $S$  is data-refined by  $T$  under abstraction invariant  $AI$ ’. Data refinement is defined as follows [1]:

**Rule 3.1 (Data Refinement).**  $S \sqsubseteq_{AI} T$  if the following holds<sup>3</sup>:

$$AI \Rightarrow [T](\langle S \rangle AI).$$

<sup>2</sup>  $[S]P$  is equivalent to Dijkstra’s  $wp(S, P)$  [9].

<sup>3</sup> It is also required that  $[S]true$ , the *precondition* of  $S$ , holds, but we ignore this since we only use statements whose preconditions are *true* in this paper.

```

MACHINE FileTransferRefinement
REFINES FileTransfer
CONJOINS FileTransferRefinementActs(Byte)

INVARIANT
  (ByteWise ∈ {BW1, BW2} ⇒ g = afile) ∧
  (ByteWise = Transfer ⇒ g = bfile ∧ afile) ∧
  (ByteWise = Transfer1 ⇒ g = bfile)
  ∧
  (Copy = Idle ⇔ ByteWise ∈ {BW, Transfer2}) ∧
  (Copy = Remember ⇔
    ByteWise ∈ {BW1, BW2, Transfer, Transfer1})

ALPHABET Send(f : File)  f ← Receive
          OpenReq OpenResp TransBlock EndTrans Ack

PROCESS ByteWise = BW WHERE
  BW = Send → OpenReq → OpenResp → Transfer
  Transfer =
    IF afile = []
    THEN EndTrans → Receive → Ack → BW
    ELSE TransBlock → Transfer

END
END

```

**Fig. 10.** CSP part of *FileTransfer* refinement.

**Rule 3.2 (System Refinement I).** A system  $M$  is refined by system  $N$  under abstraction invariant  $AI$  if  $N$  has an action  $N.a$  corresponding to each action  $M.a$  and for each such  $a$ :

$$M.a \sqsubseteq_{AI} N.a.$$

Furthermore, the initialisation of  $M$  must be data refined by the initialisation of  $N$ .

### 3.1. Refinements in csp2B

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.

Figs. 10 and 11 give an example of a refinement of this form. The CSP part is described in Fig. 10 and this specification is conjoined with the B machine of Fig. 11. This is a refinement of the *FileTransfer* machine of Fig. 3 and it transfers

```

MACHINE FileTransferRefinementActs(Byte)
VARIABLES afile, bfile
INVARIANT afile ∈ File ∧ bfile ∈ File

OPERATIONS

  Send_Act(f) ≐ PRE f ∈ File THEN afile := f END;

  f ← Receive_Act ≐ f := bfile;

  OpenResp_Act ≐ bfile := [];

  TransBlock_Act ≐ bfile := bfile ∧ (first afile) || afile := tail(afile)

END

```

**Fig. 11.** B part of *FileTransfer* refinement.

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 abstraction invariant used to prove the refinement is included in the CSP specification of Fig. 10. 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.

#### 4. Semantic Issues

In [12], 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 [14]. 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 [18] 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 ma-

chine (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.

#### 4.1. 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,

$$P = Q_1 \parallel \cdots \parallel Q_n ,$$

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.

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.

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

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

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

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

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

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

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:

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

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

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

becomes

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

## 4.2. 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  $a.i.j.k$  and an event of the form  $a.i?y!k$  stands for the set of labels  $\{ a.i.j.k \mid j \in Y \}$ , 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

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

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

$$I(v) = \dots \parallel \text{IF } G(v) \text{ THEN } a.i?y!k \rightarrow I'(V) \parallel \dots, \quad (2)$$

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

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

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

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:

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

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  $S \parallel T$ .

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. We make use of the notion of conjugate weakest precondition from Section 3. If the machine contains an operation of the form

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

then the transition

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

is possible in state  $I(w)$  provided

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

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.

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:

$$\langle S \parallel T \rangle Q = \langle S \rangle Q \vee \langle T \rangle Q . \quad (6)$$

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.

### 4.3. 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  $\Sigma$  and  $i$  ranges over an indexing set  $I$ , then the state

space of the large LTS is  $I \rightarrow \Sigma$ . This is the basis for the translation to B of interleaving described in Section 2.4.

#### 4.4. 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:

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

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:

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

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  $S \parallel T$  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

$$\langle S \parallel T \rangle (M \wedge N) = \langle S \rangle M \wedge \langle T \rangle N. \quad (8)$$

Using this result, it is easy to show that the relationship between transitions allowed by  $S$  and  $T$  and those allowed by  $S \parallel T$  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.

Refinement of CSP processes is defined in terms of the failures-divergences model. Based on [15, 19], [6] defines the Failures-Divergences semantics of B machines and shows that refinement of B machines corresponds to refinement at the failures-divergences level<sup>4</sup>. Roscoe [18] describes the relationship between

---

<sup>4</sup> 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.

an operational semantics of CSP and a failures-divergence semantics. Using this relationship he shows that the operational definition of the CSP parallel operator is consistent with the failures-divergence definition. Since our operational definition of parallelism is equivalent to Roscoe's in the absence of variable sharing, we can claim that our parallelism is consistent with the failures-divergence definition. 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.

#### 4.5. Divergence and Nontermination

The hiding operator of CSP is used to interalise events so that they are outside the control of the environment of a process. If a process is in a state where some internal events can be executed infinitely often then the process is said to diverge. 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 [15] 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`. Recall the difference between `PRE` and `SELECT` statements: a `PRE` statement aborts (i.e., is not guaranteed to terminate) when  $Q$  is not satisfied, while a `SELECT` statement is disabled and hence cannot be executed when  $Q$  is not satisfied.

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 [15]. (We have found this latter approach to be the most convenient.) 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:

$$S \parallel \text{SELECT } \textit{grd}(S) \text{ THEN } T \text{ END} .$$

If  $T$  was not guarded by  $\textit{grd}(S)$ , 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  $S \parallel T$ , and we have, for example, that

$$\text{SELECT } \textit{false} \text{ THEN } \textit{skip} \text{ END} \parallel \textit{abort} = \textit{abort} .$$

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.

## 4.6. 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 than 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 [1] 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.

## 5. 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.

The approach of  $SL_0$  and CSP-OZ mentioned in the introduction is related to csp2B though, unlike csp2B, it does not allow event ordering to be used when refining the state part. The idea of modelling a program counter more abstractly has been used in action systems elsewhere [17]. There, a binary sequencing operator is introduced which is really a shorthand for the appropriate use of a program counter. This allows for less general sequencing constraints than our approach. In Lamport's TLA [13], a state machine is constrained by temporal logic formulae rather than event-ordering terms and rules are provided for eliminating these in refinement. Abrial and Mussat [2] also provide rules for checking that a B machine satisfies simple temporal properties.

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

$$(i \rightarrow P) \parallel (i \rightarrow Q),$$

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 [6]. The tool has been applied to a larger example (a form of distributed database) [11] 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] R.J.R. Back and K. Sere. Stepwise refinement of parallel algorithms. *Science of Computer Programming*, 13:133–180, 1989.
- [5] M.J. Butler. Stepwise refinement of communicating systems. *Science of Computer Programming*, 27(2):139–173, September 1996.
- [6] 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.
- [7] M.J. Butler. csp2B User Manual. Available from <http://www.ecs.soton.ac.uk/~mjb/csp2B>, 1999.
- [8] M.J. Butler and M. Waldén. Distributed system development in B. In H. Habrias, editor, *First B Conference*, November 1996.
- [9] E.W. Dijkstra. *A Discipline of Programming*. Prentice–Hall, 1976.
- [10] C. Fischer. CSP-OZ: A combination of Object-Z and CSP. In H. Bowmann and J. Derrick, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS'97)*, pages 423–438. Chapman & Hall, 1997.
- [11] 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>.
- [12] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice–Hall, 1985.
- [13] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.
- [14] R. Milner. *Communication and Concurrency*. Prentice–Hall, 1989.
- [15] 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.
- [16] E.-R. Olderog. Towards a design calculus for communicating programs. In J.C.M. Baeten and J.F. Groote, editors, *Proceedings CONCUR '91*, volume LNCS 527. Springer–Verlag, 1991.
- [17] J. Plosila, R. Ruksenas, and K. Sere. *Action Systems Synthesis of DI Circuits*. Technical Report 149, <http://www.tucs.abo.fi/publications/techreports/TR149.html>, Turku Centre for Computer Science (TUCS), December 1997.
- [18] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice–Hall, 1998.
- [19] 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.