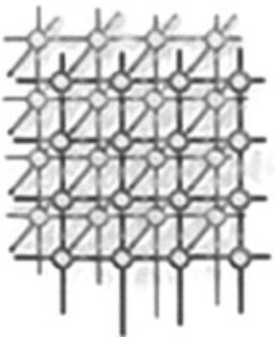
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2010; **22**:1007–1022
Published online 29 October 2009 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.1425



Java implementation platform for the integrated state- and event-based specification in ProB‡

L. Yang and M. R. Poppleton*, †

Dependable Systems and Software Engineering Group, Electronics and Computer Science, University of Southampton, University Road, Southampton SO17 1BJ, U.K.

SUMMARY

ProB is an animation and model checking tool, which supports integrated event- and state-based specifications combining B and CSP. We present an initial strategy for implementing the combined specification model as a concurrent Java program. Our Java implementation for the combined B and CSP model uses a similar approach to that of JCSP. The restricted operational semantics for the integrated B and CSP model in ProB is defined. Then a new Java package, JCSProB, is developed for implementing the semantics. The new package supports external choice with multi-way synchronization, and introduces an improved multi-threading implementation from JCSP. Copyright © 2009 John Wiley & Sons, Ltd.

Received 4 February 2009; Accepted 10 February 2009

KEY WORDS: B-method; concurrency; CSP; formal methods; JCSP; ProB

1. INTRODUCTION

Formal approaches to modelling and developing concurrent computer systems, such as CSP [1] and CCS [2], have been in existence for more than thirty years. Many research projects and a number of real world systems [3–5] have been developed from them. However, most programming languages in the industry, which support concurrency, still lack formally defined concurrency models to make the development of such systems more reliable and tractable. Liveness and fairness issues, such as

*Correspondence to: M. R. Poppleton, Dependable Systems & Software Engineering Group, Electronics and Computer Science, University of Southampton, University Road, Southampton SO17 1BJ, U.K.
†E-mail: mrp@ecs.soton.ac.uk

‡Revised version of Yang L, Poppleton MR. JCSProB: A Java implementation of the integrated specification in ProB. In McEwan AA, Schneider S, Ifill W, Welch PH (eds). Communicating Process Architectures 2007. IOS Press: Amsterdam. Published with permission from IOS Press.

deadlock and starvation, are usually intractable, and depend totally on the developers' skills and experience in concurrent systems development. Therefore, many approaches have been attempted to formalize the development of concurrent Java systems.

Formal analysis techniques have been applied to concurrent Java programs. JML [6] and Jassda [7] provide strategies to add assertions to Java programs, and employ runtime verification techniques to check the assertions. Such approaches are concerned with the satisfaction of assertions in Java code, not explicit verification against a formal concurrency model. An explicit formal concurrency model, which can be verifiably transformed into a concurrent Java program, would represent a useful contribution.

Magee and Kramer [8] introduce a process algebra language, finite state processes, and provide a formal concurrency model for developing concurrent Java programs. Then the labelled transition system analyser tool is employed to translate the formal model into a graphical equivalent. However, as no mapping has been defined from the formal model to Java, a Java implementation, here, is a manual transcription from the formal model. There is no guarantee that the Java code would be a correct implementation of the formal model.

JCSP [9] is a Java implementation of the CSP/*occam* language. It implements the main CSP/*occam* structures, such as process and channel, as well as key CSP/*occam* concurrency features, such as parallel, external choice and sequential composition, in various Java interfaces and classes. It bridges the gap between specification and implementation. With all the Java facility components in the JCSP package, developers can easily construct a concurrent Java program from its CSP/*occam* specification. The correctness of the JCSP translation of the *occam* channel to a JCSP channel class has been formally proved [10]: the CSP model of the JCSP channel communication was shown to refine the CSP/*occam* concurrency model. Early versions of JCSP (before 1.0rc6) targeted classical *occam*, which only supported point-to-point communication, while recently, new versions of JCSP have moved on to support the *occam*-π language, which extends the classical *occam* with π-calculus. More CSP mechanisms, e.g. external choice over multi-way synchronization, have been implemented in JCSP version 1.0rc7. Our work is mainly based on JCSP 1.0rc5, while we plan to move to 1.0rc7. We will discuss this in Section 5.

Raju *et al.* [11] developed a tool to translate the *occam* subset of CSP/*occam* directly into Java with the JCSP package. Although in our experience the tool is not robust enough to handle complex examples, it provides a useful attempt at building automatic tool support for the JCSP package.

Recent research on integrating state- and event- based formal approaches has been widely recognized as a promising trend in modelling large-scale systems [12–15]. State-based specification is appropriate when data structure and its atomic transition is relatively complex; event-based specification is preferred when design complexity lies in behaviour, i.e. event and action sequencing and synchronization between concurrent or distributed processes. In general of course, significant systems will present design complexity, and consequently require rich modeling capabilities, in both aspects. CSP-Oz [16], csp2B [13], CSP $\|$ B [14] and Circus [15] are all existing integrated formal approaches. However, the lack of direct tool support is a major limitation of these approaches. Proving the correctness of their combined specifications requires complex techniques, such as composing the verification results from different verification tools [17], or translating the combined specification back into a single specification language [13,18].

DOI: 10.1002/cpe

The PROB tool [19] is designed for animation and model checking formal models. It supports an integrated formal approach [20], which combines B [21] and CSP[§]. A composite specification in PROB uses B for data definition and operations. For a concurrent system, a CSP specification is employed as a filter on the invocations of atomic B operations, thus guiding their execution sequence. An operational semantics in [20] provides the formal basis for combining the two specifications. The PROB tool, which was designed for the classical B method, provides invariant checking, trace and singleton-failure refinement checking and is able to detect deadlock in the state space of the combined model.

The main issue in developing an implementation strategy for PROB is how to implement the concurrency model of the B+CSP specification in a correct and straightforward way. Furthermore, we need an explicit formal definition, and ideally, automatic tool support, to close the gap between the abstract specification and concrete programming languages. The structure of the JCSP package gives significant inspiration. We have implemented the B+CSP concurrency model as a new Java package, JCSProB, with a process–channel structure similar to JCSP. In our previous work [22,23], we formally defined a set of translation rules to convert a useful and deterministic subset of B+CSP specification to Java code. To make the translation more effective and stable, an automatic translation tool was constructed as a functional component of the PROB tool. This paper is an extended and revised version of that presented in [23].

In this paper we define a restricted operational semantics for the combined B+CSP model of PROB. The restricted semantics reduces non-determinism in the original B+CSP model. It is concrete enough to be implemented into Java programs, and abstract enough to support modelling most general systems. In Section 2 we introduce the combined B+CSP specification, and our restrictions on its semantics.

Our second contribution is the Java implementation strategy for the B+CSP concurrency model. It implements basic features of the combined abstract specification, and provides the fundamental components for constructing concurrent Java programs. We discuss the development of the JCSProB implementation package, and how it differs from that of JCSP, in Section 3. A new thread/process mechanism is introduced to support process calls and mutual recursion. In Section 4, we use an example to demonstrate these developments in JCSProB and in the translation. Section 5 concludes, discussing ongoing work, including GUI development and scalability issues.

2. THE COMBINED B+CSP SPECIFICATION

As our work is inspired by the development of JCSP, when we discuss the Java implementation in this section, we compare it with JCSP in the various aspects. We first give a brief introduction to the B+CSP specification. Then we discuss the operational semantics of B+CSP, and the restricted semantics used in this work. Finally, we demonstrate how the semantics works.

The B part of the combined specification language supported in our approach is mainly from the B0 subset. B0 is the concrete, deterministic subset of the B language describing operations and data of implementations. It is designed to be mechanically translatable to programming languages such

[§] We will call this notation B+CSP for shorthand.

DOI: 10.1002/cpe

```
MACHINE lift
VARIABLES level
INVARIANT level : NAT & level ≥ 0 & level ≤ 10
INITIALIZATION level := 1
OPERATIONS
    inc = PRE level < 10 THEN level := level + 1 END;
    dec = PRE level > 0 THEN level := level - 1 END
END
```

Figure 1. An example B machine: the lift.

as C and Ada. A B machine defines data variables in the **VARIABLES** clause, and data substitutions in the **OPERATIONS** clause. Possibly subject to a **PRE**condition—all of whose clauses must be satisfied to enable the operation—an operation updates system state using various forms of data substitution. Although the B specification used in our approach is the B0 subset, we do support some abstract B features, which are not in B0, e.g. precondition. These features are implemented to provide extra capability for rapidly implementing and testing more abstract specifications in Java programs. In the implementation, preconditions are interpreted as guards, which will block the process if the precondition is not satisfied.

A B operation may have input and/or output arguments. For an operation *op* with a header $rr \leftarrow op(ii)$, *ii* is a list of input arguments to the operation, while *rr* is a list of return arguments from it. The **INITIALISATION** clause establishes the initial state of the system. The **INVARIANT** clause specifies the safety properties on the data variables. These properties must be preserved for all the system states. Figure 1 shows a simple lift example in a B machine. It has a variable *level*, which indicates the level of the lift, and two operations, *inc* and *dec* to move the lift up and down.

Turning to the integrated specification, currently ProB only supports one paired B and CSP combination. Although ProB supports trace refinement checking for the combined specification, it does not, at present, provide a refinement strategy for composing or decomposing an abstract B+CSP model into a concrete distributed system. On the other hand, the CSP‖B approach does provide a composition technique [17] for composing combined B and CSP specifications. A full FDR-compliant CSP support has been recently implemented in ProB. Therefore, it is very likely that the CSP‖B approach can be connected with the ProB tool. Our work here focuses on one concrete B and CSP specification pair.

A detailed definition of supported CSP syntax can be found in the ProB tool.

2.1. The operational semantics of B+CSP

The operational semantics of B+CSP is introduced in [20]. It provides a formal basis for combining the B and CSP specification. The B and CSP specifications are composed as parallel processes. A B machine is viewed as a special process in the system, which maintains and updates the system state through the data transitions in its operations.

Without CSP processes, a B machine process can fire all its operations freely. The data transitions can only be blocked by preconditions on the operation. However, in some cases, it may not be very convenient to define system level behaviours only with precondition guards. Normally, a B machine needs to define an abstract 'program counter' and use it in preconditions to control the execution

DOI: 10.1002/cpe

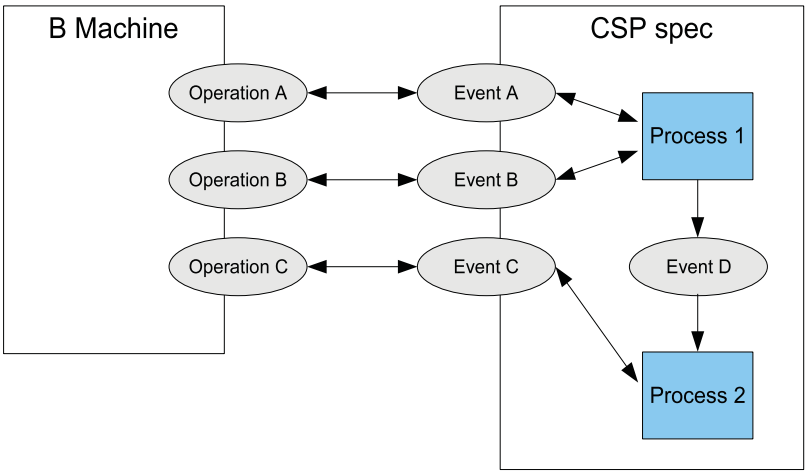


Figure 2. The synchronization between B and CSP specification.

of an operation [24]. However, this form of specification of behaviour is opaque compared with process algebra approaches such as CSP.

To work with CSP processes, a B machine process needs to synchronize and communicate. The synchronization and integration of B and CSP processes are on the B operations and CSP channels.

- A B operation must have a corresponding CSP channel with the identical name. Together, they form a combined B+CSP channel. The B operation is only ready to progress when the corresponding CSP channel is also ready.
- A CSP channel is combined with a B operation, or it can be a pure CSP channel, which has no B counterpart. A pure CSP channel is only used in the CSP part for communication.

Figure 2 illustrates how the synchronization works. Operations A, B and C all have corresponding CSP channels in the CSP part. Only when channel A is ready, operation A is able to progress the data transitions inside the operation. CSP channel D is only used by CSP processes, and has no counterpart in the B machine. In this way, the system behaviour specified in CSP can be used to control the execution of data transitions in the B machine.

The combined B+CSP event is defined by the operational semantics. A state of a combined B+CSP specification is defined as a pair, which includes a B state and a CSP state.

In [20], states σ and σ' are the before and after B states for executing a B operation. The operation is defined with operation identifier op, return variable $r_1, \ldots, r_m$, and input variables $a_1, \ldots, a_n$, as $r_1, \ldots, r_m \leftarrow op(a_1, \ldots, a_n)$. The B operational semantics can thus be defined with a ternary relation $\rightarrow$ as $\sigma \rightarrow_{op} \sigma'$. That means in state σ, the operation op progresses with input variable $a_1, \ldots, a_n$, then returns output variables $r_1, \ldots, r_m$, and reaches a new state σ'. In the CSP part, P is a CSP process, and P' is the process after P processing CSP channel ch, which has the same identical name as the B operation op. Channel ch can be defined with a number of variables $b_1, \ldots, b_i$ as $ch.b_1 \ldots . b_i$. The CSP operational semantics is give by a similar relation $\rightarrow$ as $P \rightarrow_{ch} P'$.

DOI: 10.1002/cpe

Therefore, the before and after state of B+CSP specification can be defined as (σ, P) and (σ', P'). We can now define the operational semantics of B+CSP specification by combining the two ternary relations into one form $(\sigma, P) \rightarrow_{ev} (\sigma', P')$, where the combined event *ev* is the synchronization of B operation *op* and CSP channel *ch*.

The essential issue of this synchronization is how to define the data flows of B operations and CSP channels. A B operation can have input variables $a_1, \ldots, a_n$ and return output variables $r_1, \ldots, r_m$ as result, while variables $b_1, \ldots, b_i$ of a CSP channel can have input(?), output(!) and dot(.) decorations to imply the data flow of the variables.

ProB supports a very flexible way to combine the data flows of B operations and CSP channels. In ProB, the synchronization is achieved by Prolog unification, which means data information can flow in from both B and CSP:

- CSP channels can provide concrete data values, which means the CSP part is used to drive the B part. For example, a CSP channel $ch!a$ can output data a to drive a corresponding B operation $op(a)$.
- B operations can provide data values, which means that the B part is used to drive the CSP part. For example, a B operation $b \leftarrow op$ can provide data for the corresponding CSP channel $ch?b$.
- B and CSP can both provide concrete data values to each other. The mixed data flow allows B and CSP can drive each other at the same time. In this case, a CSP channel $ch!a?b$ and a B operation $b \leftarrow op(a)$.
- In the worst case, when both B and CSP do not provide concrete data values, ProB can enumerate the B datatypes of variables and drive the interpreter.

As an abstract model checking tool, ProB tries to explore all the possible states of a system. The power of enumerating data values from datatype definitions makes it capable of using the combined channels without caring about the input/output data flow on the channels. Therefore, it does not clearly distinguish the input and output variables of both B and CSP.

2.2. The restricted semantics of B+CSP

As a model checking tool, ProB aims to exhaustively explore all the states of an abstract finite state system, on the way enumerating all possible value combinations of operation arguments. The flexibility in combining the two formal models provides more power to the ProB tool to model check the state space of a model. The implementation of the semantics using Prolog unification is simple and efficient. In some cases, it allows the ProB interpreter, instead the B or the CSP, to drive the combined model. However, as our target is implementing the combined B+CSP specification in a concrete programming language, we cannot implement the involvement of the ProB interpreter or support the same flexible and abstract semantics as model checkers. Therefore, we have to restrict the original semantics in ProB to make it suitable and meaningful for a concrete programming language.

The ProB interpreter is used when neither the B nor the CSP provides full data information to drive the model. There are three kinds of cases, where this can happen:

- Both the B operation $op(aa)$ and the CSP channel $ch?aa$ request the data value of variable aa.
- The B operation $op(aa)$ requests the data value of variable aa, while CSP channel ch does not provide the value.

DOI: 10.1002/cpe

Table I. The allowed arguments combination for pure CSP event.

JCSP	CSP input ($c?y$)	CSP output ($c!x$)	CSP none (c)
CSP output ($c!x$)	$\surd$ (p2p sync)	×	×
CSP input ($c?y$)	×	$\surd$ (p2p sync)	×
CSP none (c)	×	×	$\surd$ (barrier)

- The CSP channel *ch*?*aa* requests the data value of variable *aa*, while B operation *op* does not provide the value.

In our restricted semantics, we prohibit all the three combinations. Furthermore, there is another combination of the B and CSP variables dropped from the semantics.

- Both the B operation $aa \leftarrow op$ and the CSP channel *ch*!*aa* output data values.

ProB can handle this with its Prolog unification. Only when the two output values from the B and the CSP are the same, can the B operation and the CSP channel be combined. However, this violates the concurrency model of our approach.

We thus define a restricted B+CSP operational semantics as follows. For a B operation $o_1, \ldots, o_m \leftarrow op(i_1, \ldots, i_n)$, its corresponding CSP channel must be in the form of $ch!i_1 \ldots !i_n?o_1 \ldots ?o_m$. At CSP state P, a CSP process sends channel arguments $i_1, \ldots, i_n$ through the channel to a B operation as input arguments. After the data transitions of the channel complete—taking B state from σ to σ'—the CSP state changes to P'. The arguments $o_1, \ldots, o_m$ represent the data returned from B to CSP. The input arguments $i_1, \ldots, i_n$ only exist in state (σ, P), while the output arguments $o_1, \ldots, o_m$ are only available in state (σ', P'). The new restricted semantics of a combined event *ev* can be expressed as $((\sigma, P), in) \rightarrow_{ev} ((\sigma', P'), out)$, where $in = i_1, \ldots, i_n$, and $out = o_1, \ldots, o_m$.

ProB also supports the classical CSP communication channels. These channels exist only in the CSP part of the combined specification and have no B counterparts, which means that they cannot directly affect the system states in the B part. A channel output ($c!y$) synchronizes with some corresponding channel input ($c?x$) from a different process, and transfers a data item. This synchronization is a point-to-point(p2p) communication pattern. It also supports multi-way synchronization, multiple processes can synchronize on one barrier channel c. Table I demonstrates CSP communication channels supported in this work.

3. THE JAVA IMPLEMENTATION OF B+CSP

The JCSP package enables the implementation of CSP/*occam* formal specifications in Java. Our combined B+CSP specifications are expressed in a much larger language than the classical *occam* subset of CSP. Although the *occam*-π language extended the *occam* language and supports multi-way synchronization, its semantics are still different from that of B+CSP. However, it is possible to use *occam*-π to express the semantics of B+CSP. That means that it is also possible to use the new JCSP package to construct the implementation of B+CSP.

DOI: 10.1002/cpe

Table II. Comparing Java classes from JCSP and JCSProB.

CSP	JCSP	JCSProB
CSP process	The *CSProcess* class	The *BCSProcess* class
Process container	The *ParThread* class	The *RecurThread* class
Parallel composition	The *Parallel* class	The *CSParallel* class
Sequential composition	The *Sequence* class	The *CSPSequence* class
External choice guard	The *Guard* class	The *BCSPGuard* class
External choice	The *Alternative* class	The *Alter* class
CSP event	JCSP channel classes	JCSProB event classes

In the previous JCSP packages before 1.0rc7, there were no facilities for multi-way synchronization on external choice (*AltingBarrier*), or atomic state change during an extended rendezvous. Like *occam-π*, these versions implement a *barrier* class, which supports the synchronization of more than two processes. However, the *barrier* class is not embedded with the guards for external choice.

State change is the other issue of concern. JCSP channels are mainly used for communication and synchronization. The state change can only happen in JCSP process objects, while in B+CSP, only the B part of combined events can access the system variables and change the system state. Therefore, we need to implement the data transitions on system states inside the implementation of combined events.

To deal with these limitations, we construct a new Java package, JCSProB, to implement the B+CSP semantics and concurrency. This package provides infrastructure for constructing concurrent Java programs from B+CSP specifications. Different Java classes provided by JCSP and JCSProB are compared in Table II.

3.1. Event classes

The channel class in JCSProB is *PCChannel*; all the channel classes in the Java application need to extend this class to obtain the implemented B+CSP semantics and concurrency. The data transitions of a channel should be implemented in the *run()* method of the channel class.

The allowed argument combinations for the restricted semantics are shown in Table III. The *PCChannel* class provides four methods to implement this semantics policy. All the input and output arguments are grouped into objects of Java *Vector* class (*java.util.Vector*):

- ***void ready()***: there is no input/output on the combined channel
- ***void ready(Vector InputVec)***: CSP process passes arguments to B operation
- ***Vector ready_rtn()***: CSP process receives arguments from B operation
- ***Vector ready_rtn(Vector InputVec)***: CSP process passes arguments to B operation, and receives arguments from B operation.

Implementing synchronization in the restricted B+CSP concurrency is another important task for the *PCChannel* class. When there is more than one process synchronizing on a channel,

DOI: 10.1002/cpe

Table III. The allowed arguments combination for B+CSP events.

JCSProB	B: input arguments ($c(x)$)	B: return arguments ($y \leftarrow c$)	B: no argument (c)
CSP output ($c!x, c.x$)	$\surd$	$\times$	$\times$
CSP input ($c?y$)	$\times$	$\surd$	$\times$
CSP none (c)	$\times$	$\times$	$\surd$

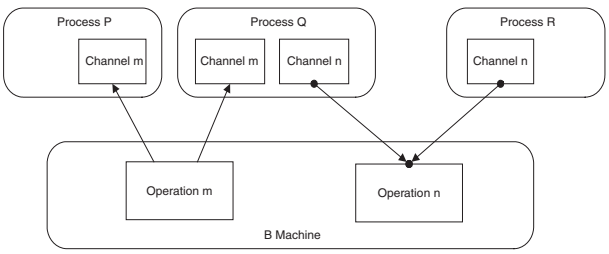


Figure 3. The synchronization between B and CSP specification.

the *run()* method will not be invoked until the condition from the concurrency model is satisfied. In the *PCChannel* class, we implement the synchronization illustrated in Section 2.2. The *inc_syn_procs_no(int)* method from *PCChannel* class is used to indicate the number of processes, which synchronize on it. For example, in Figure 3 the *inc_syn_procs_no(int)* method of channel *n* is called to indicate that process *Q* and *R* synchronize on this channel, before the two processes are initialized in the *MAIN* process.

The following Java code shows how this mechanism is implemented:

```
n_ch.inc_syn_procs_no(2);
new Parallel(
      new CSProcess[]{
            new P_procclass(var,m_ch),
            new Q_procclass(var,m_ch,n_ch),
            new R_procclass(var,n_ch),
      }
).run();
```

Process classes *P_procclass*, *Q_procclass* and *R_procclass* are running in parallel. An instance of *Parallel* class from JCSP package groups all three of them together, and uses the *run()* method to run the three processes in parallel. The *inc_syn_procs_no(int)* method of channel object *n_ch* is called to inform the channel that there are two processes, *Q_procclass* and *R_procclass*, synchronizing on it. Although channel object *m_ch* is also shared by two processes *P_procclass*, *Q_procclass*, the two processes interleave with each other, and do not synchronize on it.

There are two other issues concerning the *PCChannel* class. One is the precondition check, which can guard conditions on the data transitions inside a B operation. The *PCChannel* class provides a

DOI: 10.1002/cpe

method *preConditionCheck()* for checking the precondition on the data transition, and blocking the caller process when the condition is not satisfied. The actual precondition should be implemented in the *preCondition()* method of the channel class. The default *preCondition()* method in *PCChannel* guards on no condition, and always indicates that the precondition is satisfied. The concrete channel subclass needs to override the *preCondition()* method to implement the precondition.

The other issue is the implementation of atomic access by the B operations. The JCSProB packages provide a *JcspVar* class for implementing this feature in the Java implementation. It explicitly implements an exclusive lock to control the access to the B variables. Only one channel object can have the lock at a time. When a subclass of *PCChannel* tries to override the *run()* method, it is forced to use method *lock()* from *JcspVar* class to obtain the access authorization first, and release it by calling method *unlock()* after data transitions. When constructing a Java implementation from its formal specification, the *JcspVar* class needs to be extended, and all the global B variables should be implemented in the new constructed class.

3.2. Process and thread classes

A notable difference between B+CSP and JCSP/*occam-π* is how they implement recursion in the process. *occam-π* uses a `WHILE` loop structure in a process to implement recursion. The statements inside the structure are repeatedly processed if the loop condition is satisfied. In JCSP, this loop structure is implemented by the Java ***while*** loop statement, which has the same semantics as the `WHILE` loop in *occam-π*. The loop steps in a recursion take place in a single JCSP process object, without introducing any new process objects. For example, a CSP process

$P = a?x \rightarrow b!x \rightarrow P$

would be translated into a JCSP process class as:

```
class P implements CSProcess{
      ......
      void run(){
            while(true){
                  x = a.read();
                  b.write(x);
      }}
}
```

The CSP part of the B+CSP specification supports more general forms of recursion through process calls. To define a linear recursion process P in CSP, the process name P appears in some branches of P's behaviour. Semantically, after process P performs events a and b, it enters a state P'. At state P', it calls process P. Then the process would perform as a new P again. In this way, events a and b are repeatedly called. Although the linear recursion described here can be easily implemented by the loop structures in JCSP and *occam-π*, they are not the same semantically: in JCSP/*occam-π*, the loop is inside a process, which means that the loop structure maintains the state variables of the process, whereas each new process instance created in CSP carries its own process state.

DOI: 10.1002/cpe

Furthermore, using branching structures, such as condition and choice, CSP can produce more complex and mutual recursion patterns than can be modelled by the ***while*** loop in JCSP. One process can have different descendant processes depending on condition and choice, for example:

$$Q = c \rightarrow R \,\Box\, d \rightarrow Q$$

After resolving the external choice, the process Q here can perform as either Q or R. It is not clear whether it is possible to devise generic models in Java for such mutual recursion structures using simple loops.

If it is not possible to devise such a model, it would require the existing process object to declare a new process object and call its *run* method. But it is not safe to directly call a new process object in JCSP. For example, the process P above would be translated into a JCSP process class as:

```
class P implements CSProcess{
      ......
      void run(){
            x = a.read();
            b.write(x);
            new P(a,b).run();
      }
}
```

The translation is not semantically correct: the process state should not be retained when new process starts in CSP, whereas the new process object is stacked atop the existing process object, which means that the existing one cannot be released. It is also dangerous to do this in JCSP because doing so may eventually cause a Java stack overflow error (JDK has a limitation on the number of objects it can handle).

In our implementation of process calls and recursions, we replace the caller process object P with the callee process object P' in the same Java thread container. There is a synchronization barrier in a thread, which is used to inform the environment about the termination of this thread. The JCSP package has a package-private thread class *ParThread* for running JCSP process objects. The *ParThread* class only appears in the *Parallel* class, which implements the parallel composition of processes. It is not allowed to be accessed from the outside of the JCSP package.

The implementation of the CSP process in JCSProB consists of two parts: an abstract process class *BCSProcess*, which implements the *CSProcess* interface of JCSP, and a thread class *RecurThread* which is the thread container for the new process class. Every *BCSProcess* process object needs to run in its own thread container, which is an instance of the *RecurThread* class. Even for the main process, the Java program needs to produce an extra thread contrainer to run it. The *RecurThread* class consists of two fields: one is the *BCSProcess* process object which runs in it, and the other is a JCSP *Barrier* object which needs to synchronize before the thread terminates. When the thread starts running, it calls the *run* method of the process object. The *BCSProcess* class provides a *callNextProc* method for its implementation classes. When an existing process object P calls a new process object P', it calls the *callNextProc* method with P'. When P terminates, the thread

DOI: 10.1002/cpe

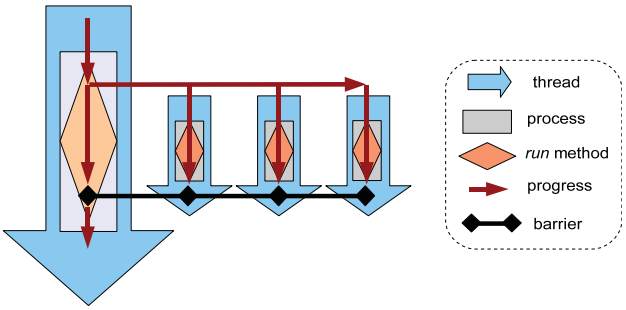


Figure 4. Parallel composition in JCSProB.

container releases the process object P, and runs the new process object P'. Therefore, the above example is implemented in JCSProB as:

```
class P implements BCSProcess{
      ......
      void run(){
            x = a.read();
            b.write(x);
            callNextProc(new P(a,b));
      }
      ......
}
```

The new thread class *RecurThread* provides a fundamental thread implementation for the B+CSP process. As the multi-threading mechanism in JCSP does not support this new thread class, we also implemented parallel composition and sequential composition for the new thread and process classes. The parallel process composition of B+CSP is implemented in the *CSParallel* class. For a parallel composition consisting of N processes, the *CSParallel* object generates N new thread objects of the *RecurThread* class to run all the N processes. All of the new N threads, and the current thread running the parallel composition, share a barrier whose counter is set to $N+1$. When all the threads terminate, the parallel composition structure completes its run. Figure 4 illustrates a parallel composition of three processes. This new implementation of parallel composition allows the process objects in all the parallel threads to be replaced by new process objects.

3.3. Automated translation: from B+CSP to Java

In [22,23], we introduced the translation rules and the translation tool, which convert B+CSP models into Java code. The JCSProB package provides the basic facilities for constructing concurrent Java applications from B+CSP models. However, as B+CSP models and the Java code represent different abstraction levels using quite different structuring concepts, it is not easy for the user to identify the mapping between them. Therefore, manually constructing the Java

DOI: 10.1002/cpe

implementation with the package is complex, and cannot guarantee whether the Java application is correctly constructed. To close the gap, a set of translation rules is developed to provide a formal connection between the combined specification and the target Java application. The translation rules can be recursively used to generate a concurrent Java application from a B+CSP model.

The automatic translation tool is constructed as a part of the ProB tool. Our translation tool is also developed in *SICStus* Prolog, which is the implementation language for ProB. In ProB, the B+CSP specification is parsed and interpreted into Prolog terms, which express the operational semantics of the combined specification. The translation tool works in the same environment as ProB, acquires information on the combined specification from the Prolog terms, and translates the information into the Java program.

4. EXAMPLES

In this section, we demonstrate implementations of guarded external choice and multi-way synchronization with the dining philosophers example. Figure 5 shows the CSP part of the specification of the combined model. The B part is simple, comprising a small number of state variables to track the states of the philosophers. As we use this example to demonstrate synchronization and recursion implementations from the CSP part, the B machine is not discussed further here.

Using the translation tool, a Java program is generated from the combined B+CSP model. The target Java program consists of a number of process classes, event classes and a state class, which are designed for maintaining state variables. For each B+CSP event, a JCSProB event class is generated. Normally, just one instance of each event class is defined in a JCSProB program. For each named CSP process, a JCSProB process class is generated, and several objects of a process class may exist in the program at runtime. For example, five process objects from the fork process class would be declared in the Java program. Figure 6 presents the *run* method from the fork process class.

The fork process *FORK_procclass* extends the abstract process class *BCSProcess* provided by JCSProB. Similar to the JCSP process classes, the behaviour of a process is expressed in the *run* method of the process class. The CSP process *FORK* starts with an external choice on the *picksup* event with different data parameters. The external choice implementation class *Alter* of JCSProB

$$
\begin{aligned}
&MAIN = PHILS[|\{picksup,putsdown\}|]FORKS\\
&PHILS = |||x{:}0..4@PHIL(x)\\
&FORKS = |||x{:}0..4@FORK(x)\\
&PHIL(x) = picksup.x.x \rightarrow picksup.x.((x+4)\%5) \rightarrow eats \rightarrow\\
&\qquad putsdown.x.((x+4)\%5) \rightarrow putsdown.x.x \rightarrow thinks \rightarrow PHIL(x)\\
&\qquad \Box\\
&\qquad picksup.x.((x+4)\%5) \rightarrow picksup.x.x \rightarrow eats \rightarrow\\
&\qquad putsdown.x.x \rightarrow putsdown.x.((x+4)\%5) \rightarrow thinks \rightarrow PHIL(x)\\
&FORK(x) = picksup.x.x \rightarrow putsdown.x.x \rightarrow FORK(x)\\
&\qquad \Box\\
&\qquad picksup.((x+1)\%5).x \rightarrow putsdown.((x+1)\%5).x \rightarrow FORK(x)
\end{aligned}
$$

Figure 5. The dining philosophers example.

DOI: 10.1002/cpe

```
.....
public class FORK_procclass extends BCSProcess{
   .....
   public void run(){
      Vector<Vector<Object>> choiceVec = new Vector<Vector<Object>>();
      choiceVec.addElement(inputVector(
         new Object[]{(Integer)proc_index_a,(Integer)proc_index_a}));
      choiceVec.addElement(inputVector(
         new Object[]{((Integer)proc_index_a + 1)%5,(Integer)proc_index_a}));
      BCSPGuard[] in = {picksup_ch,picksup_ch};
      Alter alt = new Alter(in,choiceVec);
      switch(alt.select()){
         case 0 :
            putsdown_ch.ready(inputVector(
               new Object[]{(Integer)proc_index_a,(Integer)proc_index_a}));
            callNextProc(new FORK_procclass(picks_ch,putsdown_ch,proc_index_a));
            break;
         case 1 :
            putsdown_ch.ready(inputVector(
               new Object[]{((Integer)proc_index_a + 1)%5,(Integer)proc_index_a}));
            callNextProc(new FORK_procclass(picks_ch,putsdown_ch,proc_index_a));
            break;
      }
   }
}
```

Figure 6. The dining philosophers example: the fork class.

takes two input arguments: an array *in* of *BCSPGuard* objects which stores the first event of each choice path, and a vector which stores the input data for each of these events. The *Alter* object *alt* resolves the external choice using a multi-way synchronization algorithm [25]. A choice decision is made based on the synchronization between fork and philosopher processes, as well as the preconditions of the combined events defined in the B part. The chosen event progresses inside the *alt* object, and then the choice decision returns to the fork process. The *switch* statement uses the choice decision to guide the process to continue with the selected choice path.

In CSP, each choice branch of a *FORK* process calls a new *FORK* process instance when it finishes its run. In the Java process classes, a new process object is created, and used by the *callNextProc* method as a parameter. When the current process object finishes its run, its thread container notices that there is a new subsequent process object. The thread container releases the existing process object, and starts to run the subsequent one. In this way, the recursive run of a *FORK* process is implemented with a Java implementation of CSP process calls.

5. CONCLUSION AND FUTURE WORK

This implementation strategy has been experimentally evaluated [23]. Runtime invariant checking and user-defined assertion checking were implemented and embedded inside the Java

DOI: 10.1002/cpe

implementation. We carried out a number of experiments, implementing some concurrent formal models. The known properties from the B+CSP models were evaluated in the generated Java programs. Some unknown properties, such as user-defined fairness assertions, were tested through a runtime assertion check module. This experimental evaluation of the implementation strategy gave confidence in the work, and provided a basis for addressing problems and for further development.

Our implementation strategy is strongly related to a similar approach in the *Circus* development. In [26], a set of translation rules is developed to formally define the translation from a subset of the *Circus* language to Java programs that use JCSP. As the JCSP 1.0rc5 package only supports point-to-point communication, and does not allow state change inside the channel, the supported *Circus* language subset in the translation is very limited. In [27], an ongoing effort develops an extended channel class to support multi-way synchronization. Moreover, an automatic translation tool and a brief GUI program are constructed using these translation rules. CSP/*occam* is used to model multi-way synchronization, and then JCSP to implement that model.

In [28], the new version of JCSP package (since 1.0rc7) is introduced with some new features, such as *AltingBarrier*, output guards and rendezvous. These new features provide alternative implementations for external choice with multi-way synchronization and state changes. The potential future work is to compare our implementation with the new JCSP, and even to make the two implementations compatible.

Since the current JCSProB package implements and hides the B+CSP semantics and concurrency model inside the package, the Java application generated by the translation is clear and well structured. The disadvantage is that the implementation of the B+CSP semantics and concurrency inside JCSProB still requires a formal proof of correctness of the translation. In [10], a verification technique is applied to prove the correctness of the JCSP communication channel implementation using FDR. In future work, we hope to adopt such a technique to formally model our Java implementation and prove its correctness.

A GUI package has been developed to support user interaction and configuration for JCSProB programs. The translator now also supports an option to automatically generate Java programs with GUIs. Furthermore, the runtime assertion checking module is now implemented as part of the GUI package. The user can define assertions and assertion checking algorithms as a plug-in module for JCSProB GUI programs. We will report the GUI package in the future.

Scalability is another significant issue. The JCSProB package, as well as the translation, should be applied to bigger case studies to evaluate and improve its flexibility and scalability. Currently, only one B+CSP specification pair is allowed in ProB. A proven refinement strategy for producing a concrete B0+CSP implementation from an abstract specification, as well as a technique for composition and decomposition, is still unavailable. Therefore, the JCSProB application is now restricted on a single machine. An abstract B+CSP specification cannot currently be refined and decomposed into a distributed system. In [29], an approach for composing combined B and CSP specifications—CSP $\|$ B—is presented. Work is in progress on practically applying a similar composition technique for B+CSP.

REFERENCES

1. Hoare CAR. *Communicating Sequential Processes*. Prentice-Hall International: Englewood Cliffs, NJ, 1985.
2. Milner R. *A Calculus of Communicating Systems*. Springer: Berlin, 1980.

DOI: 10.1002/cpe

3. Guiho G, Hennebert C. SACEM software validation. *Twelfth International Conference on Software Engineering*, Nice, France, 1990.
4. Deharbe D, Gomes BG, Moreira AM. Automation of Java card component development using the B method. *ICECCS 2006*, Washington, DC, U.S.A., 2006.
5. Chartier P. ABS project, merging the best practices in software design from railway and aircraft industries. *ZB2002*, *Formal Specification and Development in Z and B*, Grenoble, France, 2002.
6. Cheon Y, Leavens GT. A runtime assertion checker for the Java modeling language (JML). *International Conference on Software Engineering Research and Practice* (*SERP '02*), Las Vegas, NE. CSREA Press: Irvine, CA, U.S.A., June 2002; 322–328.
7. Brörken M, Möller M. Jassda trace assertions: Runtime checking the dynamic of Java programs. *International Conference on Testing of Communicating Systems*, Berlin, Germany, 2002.
8. Magee J, Kramer J. *Concurrency*: *State Models and Java Programs*. Wiley: New York, 1999.
9. Welch PH, Martin JM. A CSP model for Java multithreading. *ICSE 2000*, Limerick, Ireland, 2000; 114–122.
10. Welch PH, Martin JM. Formal analysis of concurrent Java system. *Communicating Process Architectures 2000*, Canterbury, England, 2000.
11. Raju V, Rong L, Stiles GS. Automatic conversion of CSP to CTJ, JCSP, and CCSP. *Communicating Process Architectures 2003*, Enschede, Netherlands, 2003; 63–81.
12. Morgan CC. Of wp and CSP. *Beauty is Our Business*: *A Birthday Salute to Edsger W. Dijkstra*. Springer: Berlin, 1990.
13. Butler MJ. csp2B: A practical approach to combining CSP and B. *World Congress on Formal Methods*. Springer: Berlin, 1999; 490–508.
14. Treharne H, Schneider S. Using a process algebra to control B operations. *IFM 1999*, York, U.K., 1999; 437–456.
15. Woodcock JCP, Cavalcanti ALC. A concurrent language for refinement. *IWFM01*: *5th Irish Workshop in Formal Methods*, *BCS Electronic Workshops in Computing*, Dublin, Ireland, 2001.
16. Fischer C. CSP-OZ: A combination of Object-Z and CSP. *Technical Report*, Fachbereich Informatik, University of Oldenburg, 1997.
17. Schneider SA, Treharne HE, Evans N. Chunks: Component verification in CSP $\|$ B. *IFM 2005*. Springer: Berlin, 2005.
18. Fischer C, Wehrheim H. Model-checking CSP-OZ specifications with FDR. *IFM 1999*. Springer: Berlin, 1999; 315–334.
19. Leuschel M, Butler MR. ProB: A model checker for B. *FME 2003* (*Lecture Notes in Computer Science*, vol. 2805). Springer: Berlin, 2003; 855–874.
20. Butler MJ, Leuschel M. Combining CSP and B for specification and property verification. *FM 2005* (*Lecture Notes in Computer Science*, vol. 3582). Springer: Berlin, 2005; 221–236.
21. Abrial J-R. *The B-Book*: *Assigning Programs to Meanings*. Cambridge University Press: Cambridge, 1996.
22. Yang L, Poppleton MR. Automatic translation from combined B and CSP specification to Java programs. *The 7th International B Conference*, Besancon, France, 17–19 January 2007.
23. Yang L, Poppleton MR. JCSProB: A Java implementation of the integrated specification in ProB. *Communicating Process Architectures 2007* (*CPA 2007*). IOS Press: Amsterdam, Netherlands, 2007.
24. Abrial J-R, Mussat L. Introducing dynamic constraints in b. *B98* (*Lecture Notes in Computer Science*, vol. 1393). Springer: Berlin, 1998; 83–128.
25. Yang L. The automated translation of integrated formal specifications into concurrent programs. *PhD Thesis* (submitted), University of Southampton, 2008.
26. Oliveira M, Cavalcanti A. From Circus to JCSP. *ICFEM 2004*, Seattle, U.S.A., 2004; 320–340.
27. Freitas A, Cavalcanti A. Automatic translation from Circus to Java. *FM 2006*. Springer: Berlin, 2006; 115–130.
28. Welch PH, Brown N, Moores J, Chalmers K, Sputh C. Integrating and extending JCSP. *Communicating Process Architectures 2007* (*CPA 2007*). IOS Press: Amsterdam, Netherlands, 2007.
29. Schneider S, Treharne H. CSP theorems for communicating B machines. *Formal Aspect of Computing* 2005; **17**(4): 390–422.

DOI: 10.1002/cpe