Communicating Process Architectures 2007
Alistair A. McEwan, Steve Schneider, Wilson Ifill, and Peter Welch
IOS Press, 2007

JCSProB: Implementing Integrated Formal Specifications in Concurrent Java

Letu YANG and Michael R. POPPLETON

Dependable Systems and Software Engineering,
Electronics and Computer Science, University of Southampton,
Southampton, SO17 1BJ, UK.

{ly03r , mrp} @ecs.soton.ac.uk

Abstract. The ProB model checker provides tool support for an integrated formal specification approach, which combines the classical state-based B language with the event-based process algebra CSP. In this paper, we present a developing strategy for implementing such a combined ProB specification as a concurrent Java program. A Java implementation of the combined B and CSP model has been developed using a similar approach to JCSP. A set of translation rules relates the formal model to its Java implementation, and we also provide a translation tool JCSProB to automatically generate a Java program from a ProB specification.

To demonstrate and exercise the tool, several B/CSP models, varying both in syntactic structure and behavioural/concurrency properties, are translated by the tool. The models manifest the presence and absence of various safety, deadlock, and bounded fairness properties; the generated Java code is shown to faithfully reproduce them. Run-time safety and bounded fairness checking is also demonstrated. The Java programs are discussed to demonstrate our implementation of the abstract B/CSP concurrency model in Java. In conclusion we consider the effectiveness and generality of the implementation strategy.

Keywords. ProB, JCSP, Integrated formal methods, Code generator

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] have been developed from them. However, most programming languages in industry, which support concurrency, still lack formally defined concurrency models to make the development of such systems more reliable and tractable. The Java language has a painful history inasmuch as it lacks explicit and formal definitions of its concurrency model. Before Java 5.0, the JMM (Java Memory Model) didn't explicitly define the read/write order that needs to be preserved in the memory model. This confused the developers of JVMs (Java Virtual Machines). The different JVMs developed under the old JMM can represent different behaviours, and lead to different results from running the same piece of Java code. To clarify this issue, Java 5.0 and the third version of the Java language specification had to redefine a new JMM.

Although the new defined JMM addressed the safety issue previously in Java concurrency, liveness and fairness issues, such as deadlock and starvation, still remain intractable, and depend totally on 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 [4] and Jassda [5] 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, 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 [6] introduce a process algebra language, FSP (Finite State Processes), and provides a formal concurrency model for developing concurrent Java programs. Then the LTSA (Labelled Transition System Analyser) tool is employed to translate the formal model into a graphical equivalence. The tool can also check desirable and undesirable properties of the FSP model. However, there is still an obvious gap in this approach between the graphical equivalence and the Java implementation. To construct the Java application, the formal model is only provided as a guidance, while the developers still need to implement the model in Java through their own experience and skill in concurrency. That means there is no guarantee that the Java code would be a correct implementation of the formal model.

JCSP [7] 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 [8]: the CSP model of the JCSP channel communication was shown to refine the CSP/*occam* concurrency model. Early versions of JCSP (before 1.0-rc6) targetted classical *occam* which only supported point-to-point communication, while recently, new versions of JCSP have moved on to support the *occam*-pi language, which extends classical *occam* with π-calculus. More CSP mechanisms, e.g. external choice over multiway synchronization, have been implemented in new JCSP (1.0-rc7). Our work is mainly based on JCSP 1.0-rc5, while we plan to move to 1.0-rc7. We will discuss this in Section 5.

Raju et al. [9] 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 modeling large-scale systems. 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 between system elements. In general of course, significant systems will present design complexity, and consequently require rich modeling capabilities, in both aspects. CSP-OZ [10], csp2B [11], CSP$\|$B [12] and Circus [13] are all existing integrated formal approaches. However, the lack of direct tool support is a one of the most serious issues for these approaches. Proving the correctness of their combined specifications requires complex techniques, such as composing the verification results from different verification tools [14], or translating the combined specification back into a single specification language [11,15].

The implementation issue is another significant question mark over integrated formal methods. The more complex structures and semantics they commonly share usually create difficulty in developing a stepwise implementation strategy for the integrated specification. For the above integrated formal approaches, only CSP-OZ has considered the association with programming languages. The applied technique, Jassda [16], is a light-weight runtime verification approach based on the *Design-by-Contract* concept [17], and is really a verification technique, rather than an implementation strategy.

ProB [18] supports an integrated formal approach [19] which combines B [20] and CSP[1]. A composite specification in ProB uses B for data definition and operations. A CSP specification is employed as a filter on the invocations of atomic B operations, thus guiding their execution sequence. An operational semantics in [19] 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 single-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, or even 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 implement the B+CSP concurrency model as a Java package with similar process-channel structure to JCSP. Based on this implementation package, we formally define 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 is constructed as a functional component of the ProB tool. Run-time invariant checking and bounded fairness assertions checking are also implemented and embedded inside the Java implementation.

There are two main contributions of this paper. The first one 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. In Section 2 we introduce the combined B+CSP specification, and our restrictions on its semantics. We then discuss the Java implementation of the concurrency model. Several key Java classes are explained, and compared with the JCSP package. The translation rules and the tool are also presented. Section 3 discusses the translation rules that are implemented in the translation tool.

The second contribution is the experimental evaluation of this implementation strategy, discussed in section 4. We carry out a number of experiments, implementing some concurrent formal models. In order to exercise the coverage of the translation rules, these models differ syntactically, using both B and CSP elements differently. Beyond exercising the translation, there are three dimensions to the experiments:

- The models illustrate the presence and absence of various behavioural properties, including safety, deadlock freeness, and bounded fairness. ProB can be used to verify the presence or absence of a safety or deadlock freeness property. In this case, we run the translated Java to *check* the translation, and expect to see the property either manifested or not, depending on whether it is present/ absent in the model.
- In the case of properties that we think might hold in the model, or that we might not even have an opinion about - such as bounded fairness - we use the Java to *simulate* the model, using a number of diverse runs to estimate the presence or absence of the property.
- We also demonstrate a simple mechanism for generating a variety of timing and interleaving Java patterns for a given input model, and consider its utility.

This experimental evaluation of the implementation strategy gives confidence in the work, and provides a basis for addressing problems and for further development.

Finally section 5 discusses the ongoing work of this approach, including GUI development and scalability issues. A formal verification of the translation is briefly discussed as necessary future work.

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

1. 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 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 the work. Finally, we demonstrate how the semantics works.

Table 1 gives the B and CSP syntax supported in our approach. We use quote marks as well as boldface to denote BNF terminal strings.

B Machine	*Machine*	**MACHINE** *Header* *Clause_machine** **END**
	Clause_machine	... | *Clause_variables* | *Clause_invariant* | *Clause_assertions* | *Clause_initialization* | *Clause_operations* | ...
B Operation	*Clause_operations*	**OPERATIONS** *Operation*$^{+\text{“;”}}$
	Operation	*Header_operation* “=” *Level1_Substitution*
	Header_operation	[*ID*$^{+\text{“,”}}$ ←] *ID* [“(” *ID*$^{+\text{“,”}}$ “)”]
B Substitution	*Precondition*	**PRE** *Condition* **THEN** *Substitution* **END**
	Block	**BEGIN** *Substitution* **END**
	If-Then-Else	**IF***Condition* **THEN***Substitution* [**ELSIF** *Condition* **THEN** *Substitution*]* [**ELSE** *Substitution*] **END**
	Var	**VAR** *ID*$^{+\text{“,”}}$ **IN** *Substitution* **END**
	Sequence	*Substitution* “**;**” *Substitution*
	Parallel	*Substitution* || *Substitution*
	Assignment	*ID*[**(** *Expression* **)**] “**:=**” *Expression*
CSP Process and Channel	*Prefix*	*ChannelExp* → *Process*
	Sequential Composition	*Process* “**;**” *Process*
	External Choice	*Process* “**[]**” *Process*
	Alphabetical Parallel	*Process* “**[**|” Ch_List“|**]**” *Process*
	Interleaving	*Process* “|||” *Process*
	Process call	*Proc_Header*
	If-Then-Else	**if** *CSP_Condition* **then** *Process* [**else** *Process*]
	Skip	**SKIP**
	Stop	**STOP**
	ChannelExp	*ID* [*Output_Parameter**] [*Input_Parameter**]
	Output_Parameter	“!”*CSPExp* | “.”*CSPExp*
	Input_Parameter	“?”*CSPExp*

Table 1. The main B and CSP specification supported in JCSProB

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 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 functions for rapidly implementing and testing some abstract specification 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 **INITIALIZATION** 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.

```
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 of B machines: lift

Table 1 also defines the supported CSP process and channel structures. A detailed definition of supported CSP syntax can be found in the ProB tool.

Currently ProB only supports one paired B and CSP combination. Although ProB supports trace refinement checking for the combined specification, it still hasn't provided a refinement strategy for composing or decomposing an abstract B+CSP model into a concrete distributed system. The CSP||B approach does provide a refinement strategy [14] for composing combined B and CSP specifications. However, it is unlikely that this approach can be directly used in ProB. Therefore our work here focusses on one concrete B and CSP specification pair. All the processes in the CSP specification are on a local machine.

1.1. The ProB Combination of B and CSP Specification

We have seen that B is essentially an action system. The system state is shared by a number of guarded atomic actions, i.e. B operations, in the system model. The actions can change the state of the system by updating the values of system variables. Whether an action is enabled is determined by its guard, a predicate on the system state. State-based formal approaches give an explicit model of data definitions and transitions. However, as behaviour is only defined by the pattern of enablement over time of the guards, any such behaviour is only observable in a runtime trace, and not explicitly in the model syntax.

An event-based approach, on the other hand, explicitly defines the behaviours of the system. The actions in the system are regarded as stateless events, i.e. the firing of CSP channels. A process, a key concept, is defined in terms of possible behaviour sequences of those events. In CSP, traces, failures and divergences semantics are used to interpret system behaviours. Thus although event-based approaches are good at explicitly defining system behaviour, they lack strength in modelling data structure and dynamics. In event-based approaches like CSP, state is nothing more than local process data, communicated through channels or by parameter passing through processes. There is no explicit way to model system states on globally defined data. a An early integration [21] of state- and event-based system models provided the theoretical correspondence between action systems and process algebras. Many approaches [10,11,12,13,19] have been made at combining existing state- and event-based formal methods. It is clearly essential, however, to provide a semantics for any proposed combined model.

The operational semantics of the B+CSP specification is introduced in [19] and provides a formal basis for combining the B and CSP specification. The B machine can be viewed as a special process which runs in parallel with CSP processes. The system state is maintained by the B machine in that process, while CSP processes only maintains their local states and cannot directly change the system state. The execution of a B operation need to synchronize with a CSP event which has the same identical name. In this way, CSP can control the firing of B operations.

The combination of a CSP event and a corresponding B operation is based on the operational semantics. The operational semantics of the combined B+CSP channel are: (σ,P) $\rightarrow_A$ (σ',P'). σ and σ' are the before and after B states for executing B operation O, while P and P' are the before and after processes for processing CSP channel ch. The combined channel A is a unification of the CSP channel $ch.a_1, ..., a_j$ and the B operation O = o_1,...,o_m $\leftarrow$ op(i_1,...,i_n).

The operational semantics of B+CSP in ProB [19] provides a very flexible way to combine B operations and CSP channels. This flexibility is in handling the arguments on the combined channel. As a model checking tool, ProB is relatively unrestricted in combining the B operation arguments and CSP channel arguments. There is no constraint on input/output directions of the arguments. CSP processes can be used to drive the execution of B machines by providing the value of the arguments, or vice-versa. It is even possible that neither B and CSP provide values for channel arguments, or that the numbers of arguments on the combined B operations and CSP channel differ. ProB can provide the values by enumerating values from the data type of the arguments. This gives more power to ProB to explore the state space of system models. However, as our target is generating concrete programs, it is not possible to allow such flexibility in the implementation semantics.

1.2. The Restricted B+CSP Semantics for JCSProB

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. However, for concrete computer programs, it is not realistic to support the same flexible and abstract semantics as model checkers. We need a more restricted and deterministic semantic definition.

We thus define a restricted B+CSP operational semantics as follows. For a B operation o = o_1,...,o_m $\leftarrow$ op(i_1,...,i_n), its corresponding CSP channel must be in the form of $ch!i_1...!i_n?o_1...?o_m$. At CSP state P, a CSP process sends channel arguments i_1,...,i_n through the channel to a B operation. 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,...,o_m represent the data returned from B to CSP. The new restricted semantics can be expressed as (σ,P,in) $\rightarrow_A$ (σ',P',out), where in = i_1,...,i_n, and out = o_1,...,o_m.

Furthermore, the flexible ProB semantics also supports CSP-only channels without B counterparts. These channels preserve the semantics of CSP/*occam*. We handle them separately from the combined B+CSP channel, and implement them in the Java application using the JCSP package. However, the CSP semantics supported by ProB is still larger than JCSP/*occam*. The allowed argument combinations in this work are showed in Table 2, although some of them have not been fully implemented yet.

JCSProB	B: input arguments (*c*(*x*))	B: return arguments (*y* $\leftarrow$ *c*)	B: no argument (*c*)
CSP output (*c!x* , *c.x*)	$\surd$ (multi-way sync)	$\times$	$\times$
CSP input (*c?y*)	$\times$	$\surd$ (multi-way sync)	$\times$
CSP none (*c*)	$\times$	$\times$	$\surd$ (multi-way sync)
JCSP	CSP input (*c?y*)	CSP output (*c!x*)	CSP none (*c*)
CSP output (*c!x*)	$\surd$ (p2p sync)	$\times$	$\times$
CSP input (*c?y*)	$\times$	$\surd$ (p2p sync)	$\times$
CSP none (*c*)	$\times$	$\times$	$\times$

Table 2. The allowed arguments combination for B operations and CSP channels

The top half of the table shows the argument combinations for the restricted B+CSP semantics. If a CSP channel *c!x* outputs an argument *x*, the argument is combined with an input argument *x* in the corresponding B operation *c*(*x*). A return argument *y* from a B operation *y* ← *c* is combined with an input argument *y* in the corresponding CSP channel *c?y*. These two kinds of arguments provide two-way data flow between the B and CSP models:

- In B state σ, CSP passes data in CSP→B arguments to invoke the execution of a B operation with these arguments. This will change system state in the B model from σ to σ'. We can see this as the CSP model reusing a stateful computation, rather like an abstract subroutine call.
- In B state σ', the return data in CSP←B arguments returns the B state to the CSP process. This can be seen as a subroutine call to read internal state, used to influence behaviour in the CSP model.

If an invocation of a B operation requires that the arguments be fixed, synchronization on the combined B+CSP channel is not only defined by the name of the channel, but also by the value of the arguments. Two processes calling a combined B+CSP channel with different argument values cannot synchronize, because the two calls represent two different data transitions in the B model. This is multi-way CSP-out-B-in synchronization.

In a similar way, multi-way B-out-CSP-in synchronization is defined, this time on the channel name only. In this case the synchronization represents one call to the B operation, returning one result, which is read by mutiple CSP input channels.

The bottom half of the table demonstrates ProB's support of the JCSP/*occam* channel in a pure JCSP semantics. As the communication in JCSP is between two processes, a call of channel output (*c!y*) corresponds to one or more channel input call (*c?x*) from other processes. The standard channel model of JCSP/occam provides point-to-point communication between a writer and read process: synchronisation happens as a by-product, since these channels provide no buffering capacity to hold any messages.

1.3. How the Restricted Semantics Works

The CSP part of the combined specification defines the behaviours of the system model. It is used to drive the execution of the B machine. It controls system behaviour by defining the execution sequence of combined channels in CSP processes, and using the channels to fire data operations in the B model. Therefore, the execution of a combined channel is guarded by a call from CSP, as well as the B precondition on the channel.

In Figure 2, process *Q* defines system behaviour by giving the execution order of channel *m* and *n*. When process *Q* calls the execution of channel *n*, whether the call will enable the data transition in the corresponding B operation *n*, is still guarded by:

- the synchronization strategy in the CSP part. In this case, as process *Q* needs to synchronize with process *R* on channel *n*, the channel is only enabled when process *R* also calls the channel.
- the precondition on the corresponding B operation *n*.

As defined in the restricted semantics, the synchronization on a combined B+CSP channel is determined by both channel name and CSP→B arguments. Multiple processes synchronize on the execution of data transitions inside a combined channel. The combined channel performs a barrier synchronization with state changes inside the barrier. Processes *Q* and *R* synchronize on channel *n*, with arguments *X+1* and *Y* on the channel respectively. The two processes will wait, and only be invoked if *X+1 = Y*, i.e. the channel arguments match. If they do not match then the calls on channel *n* will block.

On the other hand, as discussed in Section 1.2, the B state model can use CSP←B arguments to modify CSP system behaviour. In Figure 2, the B operation *m* returns an argument

rr through the combined channel. In the CSP part, as process *P* and *Q* interleave on channel *m*, both of them can receive data from the channel but on interleaving calls. Return data from different calls on the channel represent different system states in the B model. In particular, process *Q* uses the returned argument *X* afterwards to invoke the other channel *n*. This shows B state data affecting the behaviour of the combined system model.

Figure 3 shows a very simple example of one-to-one channel communication model of JCSP/*occam*, which is also supported in our semantics. Process *P* sends data *X* through channel *c* to process *Q*. When one of the processes is ready, it needs to confirm that the process on the other side of the channel is also ready for the communication. The communication only involves one reader and one writer. JCSP/occam also support multiple writers and/or readers interleaving with each other to use a shared *any-to-any* channel. Note that the writers (respectively readers) do not synchronise with each other – only one reader with one writer.

Thus two distinct concurrency models are supported, but because of their differences in synchronization, must be treated separately in translation.

2. The Java Implementation of B+CSP

2.1. JCSP and JCSProB

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

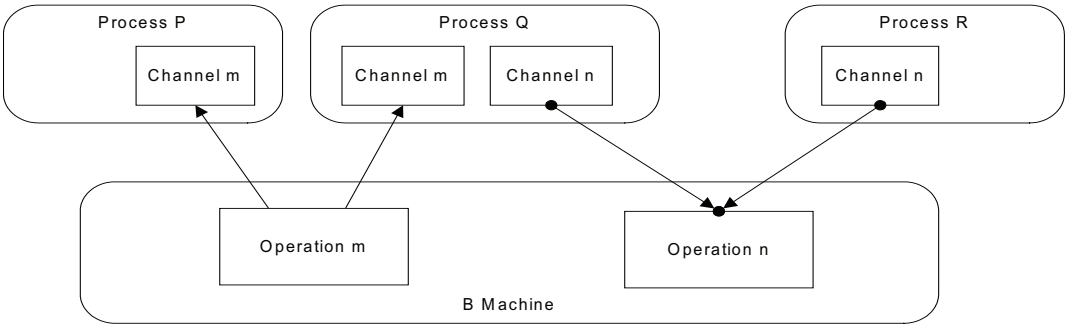
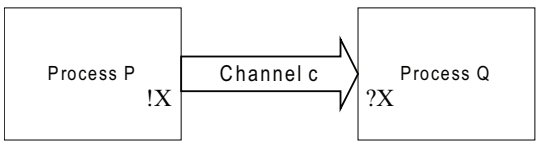


Figure 2. Concurrency Model of B+CSP in ProB



$$A = c!X \rightarrow A$$

$$B = c?X \rightarrow B$$

Figure 3. One-to-one Channel Model of JCSP

That means it is also possible to use the new JCSP package to construct the implementation of B+CSP.

When we started this work, the new JCSP package (1.0-rc7) had not been published. There were no facilities for multi-way synchronization on external choice (*AltingBarrier*), or atomic state change during an extended (rendezvous). This why we augment the point-to-point communication of previous JCSP/*occam* with a new concurrency model, called *JCSProB*. Like *occam-π*, the old JCSP package (before 1.0-rc6) implements a *barrier* class, which supports the synchronization of more than two processes. However, there is still no state change mechanism inside the *barrier* class. State change is the other issue concerned. 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 channels 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 channels.

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. In this section, we will discuss several fundamental classes from the JCSProB package. We inherit the process-channel structure from JCSP, as well using a part of its interfaces and classes. As a Java implementation of *occam* language, JCSP provides severak kinds of Java interfaces and classes:

- *CSProcess* interface, which implements the *occam* process. All the process classes in the JCSP package and in the Java application need to implement this interface.
- Some process combining classes, e.g. *Parallel*, *Sequence* and *Alternative*. They provide direct implementation of the key process structures, e.g. `PAR`, `SEQ`, and `ALT` in *occam*.
- Channel interfaces and classes. JCSP provides a set of channel interfaces and classes for implementing the point-to-point communication in *occam*.
- Barriers, alting barriers and call channels. These are not used in the work reported here, but may become useful in future developments 5.
- Timers, buckets, etc. These are not relevant here.

The JCSProB package is developed for implementing the restricted B+CSP semantics and concurrency model. In JCSProB it is mainly the channel interfaces and classes that are rewritten, as well as the process facilities which interact with the execution of channel classes, e.g. *external choice*. Figure 4 illustrates the basic structure of the JCSProB package, and its relation with JCSP, and how to build the target Java application upon these two packages.

The figure shows that there are three kinds of classes that need to be developed to construct a Java application:

- At least one process class, ⟨*process*⟩*_procclass* ("_procclass" is the suffix of a process class name), which implements the JCSP *CSProcess* interface. Each process in the CSP part of the combined specification is implemented in a process class.
- JCSProB channel classes, ⟨*channel*⟩*_chclass* ("_chclass" is the suffix of a channel class name), which extends the new *PCChannel* class from JCSProB. The *PCChannel* class implements the semantics of the combined B+CSP channel. It is an abstract class which has synchronization and precondition check mechanism implemented inside. Every channel class needs to extend this class, and override the abstract *run* method of it (If a B operation has precondition, the channel class also needs to override the *preCondition* method).
- A *MaVar* class (Machine Variable), which extends the *JcspVar* class of JCSProB. It implements the B variables, as well as the invariant and assertions on them.

The JCSP/*occam* semantics are implemented in the JCSP package. As the semantics is also supported in ProB, the JCSP package is also used in the implementation. As our approach supports both the combined semantics of B+CSP and JCSP/*occam*, the differences between the two semantics and concurrency model result in two modes of translation for two kinds of channels. B+CSP channels are translated to subclasses of *PCChannel* class from JCSProB, while JCSP/*occam* channels are translated to JCSP channel classes.

Process classes in this work implement the process interface *CSProcess* from JCSP. Some JCSP process classes, e.g. *Parallel* and *Sequence*, are also directly used to construct concurrent Java applications. Because these classes are concerned with execution orders over a set of processes, they are not concerned with internal process behaviour. For example, the *Parallel* class takes an array of process objects, and runs all the them in parallel. *Parallel* class is not involved in implementing synchronization. The synchronization strategies are implemented in the channel classes. Changing to channel classes with different synchronization strategies does not affect the functions of these process classes. Therefore, both JCSP and JCSProB channels can be used in a process class.

There is a restriction on the use of *external choice* for the two kinds of channels: B+CSP and JCSP channels cannot be used in *external choice* at the same time. The *Alternative* class from JCSP is used to implement *external choice* for JCSP channels, while *Alter* class in JCSProB implements it for JCSProB channels.

Some key JCSProB classes are discussed in the following few sections.

2.2. Channel 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 2. 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

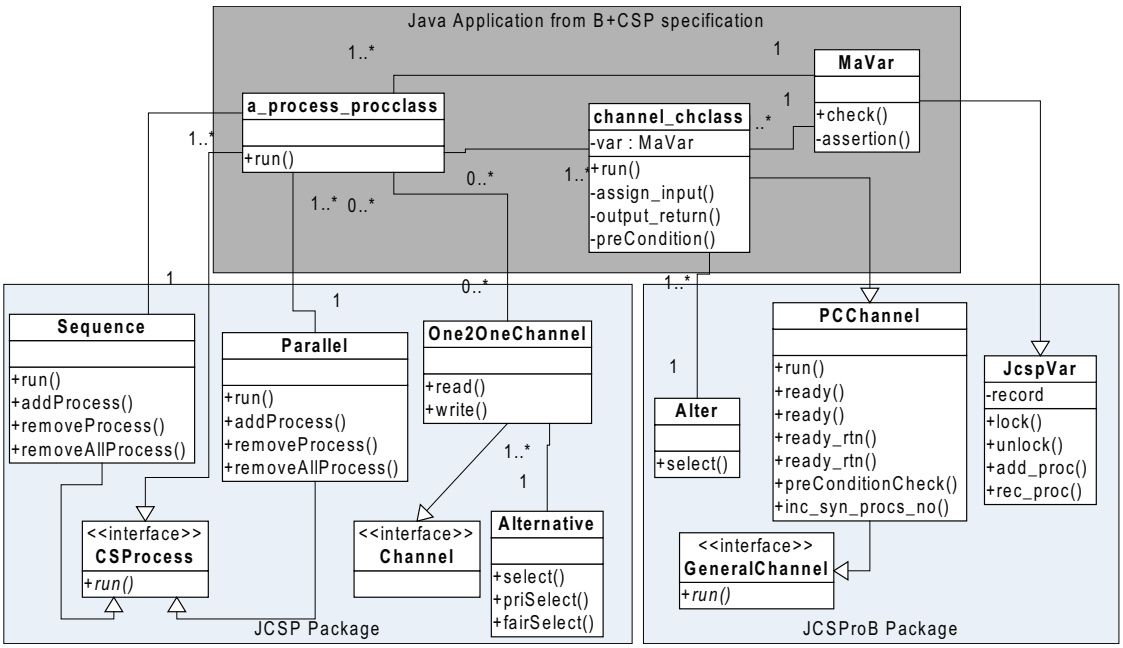


Figure 4. The structure of Java Application developed from JCSP and JCSProB packages

- ***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 the synchronization in the restricted B+CSP concurrency is another important issue for the *PCChannel* class. When there is more than one process synchronizing on a channel, 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 1.2. 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 2 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 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 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; this is discussed with the *JcspVar* class in Section 2.3.

2.3. Global B Variables Class

In B-method, the data transitions of a B operation must be kept atomic in order to preserve the consistency of the state model. The combined B+CSP model also has this requirement. 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 *lock()* method from *JcspVar* class to obtain the access authorization first, and release it by calling *unlock()* method after data transitions. When constructing a Java implementation from its formal specification, the *JcspVar* class need to be extended, and all the global B variables should be implemented in the new constructed class.

2.4. External Choice Class

In current JCSP, external choice is implemented in the *Alternative* class. As the decision of choices is based on the first events of all possible paths, *Alternative* needs to cooperate with JCSP channel objects to make the choice. Because the JCSP channel implements the *occam* point-to-point communication, the guard on the channel is based on the state of two communicating processes. In JCSP, only the guard on channel input is implemented, because guarding on channel output can cause major system overhead.

The *Alter* class from JCSProB package implements external choice for the extended channel classes *PCChannel*. It queries the *preCondition()* method of channel objects for their the preconditions, and then makes choice on the paths with ready channels. However, for the B+CSP semantics, the guard on a combined channel includes not only the precondition from the B part, but also, for the shared channel, the availabilities of all the synchronizing processes, all of whom may back off any time after offering to synchronize, having chosen something else.

Even for point-to-point communication in JCSP (1.0-rc5), the previous two-phase commit protocol for implementing guarding on channel output has been considered complex and costly. Therefore, only the guard on channel input is allowed. Implementing the guard on multi-way synchronized channel is expected to be even harder. In [22,23,24], a fast algorithm for implementing external choice with multi-way synchronization is discussed. Furthermore, for the combined B+CSP channel, the synchronization guards need to coordinate with the precondition on B operations for guarding the execution of the channel. Therefore, we are still working on multi-way synchronization guard; that work is currently under testing. We also consider using the *AltingBarrier* class in JCSP 1.0-rc7 to implement the multi-way synchronization for B+CSP channel, and it would be very interesting to compare the two implementations.

3. Translation: From B+CSP to JCSP

3.1. Translation Rules

The JCSProB package provides basic facilities for constructing concurrent Java applications from B+CSP models. However, there is still a big gap between the specification and the Java implementation. Manually constructing the Java implementation with the package is still very complex, and cannot guarantee whether the Java application is correctly constructed. To close the gap, a set of translation rules are 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.

To define the translation rules *Tr*, we first use the BNF (Backus Naur Form) notation to define the subset of B and CSP specification that can be implemented in the Java/JCSProB programs. Then the allowed target Java/JCSProB language structures are also defined in a recursive notations. With the help from a set of interpretative assumptions A, the translation rule *Tr* relates the definitions of B+CSP and Java/JCSProB:

$$B + CSP \stackrel{A}{\Longrightarrow}_{Tr} Java/JCSProB$$

The assumptions are introduced to express the B+CSP semantic features which are not obvious from the BNF-style definition. For example, in *external choice*, the channel of all possible paths can be obtained by continuously deducing the B+CSP language rules, but it is not very convenient to express them explicitly in the translation rule. Therefore, we introduce an assumption, which clearly say that a_0,...,a_n are the first channels on all the paths.

The translation rules can be classified into three parts:

- Rules for generating process classes
- Rules for generating channel classes
- Rules for generating B variable classes, invariants and assertions

In Table 3, the translation rules concerned the translation of *external choice* are listed, as well as the B+CSP syntactic structures involved. The items with angle brackets in B and CSP language specification, e.g. ⟨BOp_PreCondition⟩ , are expandable B+CSP language syntax, and the items with fat font in translation rules, e.g. $\mathbb{C}\mathrm{h}\_\mathbb{P}\mathrm{re}\mathbb{C}\_\mathbb{C}\mathrm{heck}$, are names of expandable translation rules.

CSP spec involved	⟨ProcE⟩ ⟹ ⟨ProcE:Ext_Choice⟩ | ⟨ProcE:Parallel⟩ | .... ⟨ProcE:Ext_Choice⟩ ⟹ ⟨ProcE⟩ [] ⟨ProcE⟩
B spec involved	⟨BOp_Substitution⟩ ⟹ ⟨BOp_PreCondition⟩ | ⟨BOp_Begin⟩ | .... ⟨BOp_PreCondition⟩ ⟹ **PRE** ⟨B_Condition⟩*Con* **THEN** ⟨BOp_Substitution1⟩ **END**
Additional assumptons	A1: P_0,...,P_N are all the paths for external choice A2: a_0,...,a_n are the first channels on all the external choice paths P_0,...,P_N.
Rule name	$\mathbb{P}\mathrm{roc}\mathbb{E} : \mathbb{E}\mathrm{xt}\_\mathbb{C}\mathrm{hoice}$
Rule function	Implements: ⟨ProcE:Ext_Choice⟩ Uses: ⟨ProcE⟩, ⟨BOp_Substitution⟩, A1, A2

Rule content	PCChannel[] in = $\{\mathbb{C}\mathrm{h}\_\mathbb{N}\mathrm{ame}\_\mathbb{L}\mathrm{ist}\lceil a_0, ..., a_n \rfloor\}$;
	Vector⟨Vector⟩ inputVec = new Vector⟨Vector⟩=();
	$\mathbb{E}\mathrm{c}\_\mathbb{A}\mathrm{dd}\_\mathbb{A}\mathrm{rgs}\lceil a_0, ..., a_n \rfloor$
	Alter alt = new Alter(in,choiceVec);
	switch (alt.select()){
	$\mathbb{E}\mathrm{c}\_\mathbb{C}\mathrm{hoice}\_\mathbb{P}\mathrm{aths}\lceil P_0 ... P_n \rfloor$
	}

Rule name	$\mathbb{C}\mathrm{h}\_\mathbb{P}\mathrm{re}\mathbb{C}\_\mathbb{C}\mathrm{heck}$
Rule function	Implements: ⟨BOp_PreCondition⟩

Rule content	public synchronized boolean preCondition(){
	return $\mathbb{B}\_\mathbb{C}\mathrm{onditions}\lceil Con \rfloor$;
	}

Table 3. Translation rules concerning external choice: *ProcE:Ext_Choice*

The translation rule $\mathbb{P}\mathrm{roc}\mathbb{E} : \mathbb{E}\mathrm{xt}\_\mathbb{C}\mathrm{hoice}$ is in the rule set $\mathbb{P}\mathrm{roc}\mathbb{E}$, which handles all the CSP process structures. The *rule function* indicates the B+CSP syntactical structures or assumptions that the rule implements, and uses to obtain information. The *rule content* shows the Java code that the rule generates. A very abstract lift specification in Figure 5 has an *external choice* with two paths. Note that there is a delibreate bug in the definition of the B machine in Figure 1.

The Java code in Figure 6 demonstrates how the *external choice* in the *MAIN* process is implemented in Java. Inside the rule $\mathbb{P}\mathrm{roc}\mathbb{E} : \mathbb{E}\mathrm{xt}\_\mathbb{C}\mathrm{hoice}$ a channel array which includes the first channel objects on all choice paths, is initialized first. The channel name list *inc_ch,dec_ch* is generated by rule $\mathbb{C}\mathrm{h}\_\mathbb{N}\mathrm{ame}\_\mathbb{L}\mathrm{ist}$. The Java *Vector choiceVec* stores all the argument values of the first channels of all the choices. The translation rule $\mathbb{E}\mathrm{c}\_\mathbb{A}\mathrm{dd}\_\mathbb{A}\mathrm{rgs}$ generates the Java code to add arguments on channel $a_0, ..., a_n$ to *choiceVec*. As the two channels *inc* and *dec* in the example have no argument, they just pass two null *Vector* objects to *choiceVec*.

MACHINE *lift*
VARIABLES *level*
INVARIANT *level : NAT & level* $\geq$ *0 & level* $\leq$ *10*
INITIALIZATION *level := 1*
OPERATIONS
inc = ***BEGIN*** *level := level + 1* ***END****;*
dec = ***BEGIN*** *level := level - 1* ***END***
END

———————

MAIN = inc $\longrightarrow$ *inv_check* $\longrightarrow$ *MAIN [] dec* $\longrightarrow$ *inv_check* $\longrightarrow$ *MAIN ;;*

Figure 5. Combined Specification of lift

```
PCChannel[] in = {inc_ch,dec_ch};
Vector<Vector> choiceVec = new Vector<Vector>();
{Vector<Object> inputVec = new Vector<Object>();
choiceVec.addElement(inputVec); }
{Vector<Object> inputVec = new Vector<Object>();
choiceVec.addElement(inputVec); }
Alter alt = new Alter(in,choiceVec);
switch(alt.select()){
    case 0 : { {inc_ch.ready(); }
           var.check(); }
    case 1 : { { dec_ch.ready(); }
           var.check(); }
}
```

Figure 6. Java code implementing *external choice* in the lift process class

The channel array *in*, as well as an arguments array from *choiceVec* are used to construct the *Alter* class. The *select()* method of *Alter* class chooses between the ready channel objects. Whether a channel is ready may further depend on the precondition of the B operation and the synchronization ready state, all of which depends on the argument values on that channel. Rule $\mathbb{E}\mathrm{c}\_\mathbb{C}\mathrm{hoice}\_\mathbb{P}\mathrm{aths}$ generates all the possible paths inside the choice structure. In the generated Java program, the two possible paths are represented by the two cases in the Java *switch* structure.

The implementation of external choice may also depend on the semantics implemented in other Java classes, for example, the pre-condition check. Although the precondition check mechanism is provided by the *preConditionCheck()* method of *PCChannel* class, the actual conditions are defined in the subclass of *PCChannel* through translation rules. In Table 3, translation rule $\mathbb{C}\mathrm{h}\_\mathbb{P}\mathrm{re}\mathbb{C}\_\mathbb{C}\mathrm{heck}$ is used to generate the *preCondition()* method which implement the precondition.

3.2. Translation Tool

The automatic translation tool is constructed as 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 and Experiments

In this section, experimental evaluation of the implementation strategy is discussed. We first test the usability and syntax coverage of the translation tool by using different syntactic structures to construct various formal models. Then the models are put into the translation tool. The target Java programs from different models are tested.

How the behavioural properties of the formal models are implemented in the Java programs is the other experimental target. Generally, there are two kinds of properties:

- **Known properties**. These properties, e.g. safety and deadlock, can be checked in the ProB tool for the system model. The test is whether verified properties are also preserved in the Java implementation. This provides partial evidence for the correctness of the Java implementation strategy.
- **Unknown properties**. For other properties, e.g. fairness, which cannot be verified in ProB, we provide alternative experimental means to evaluate them in the Java programs at runtime. In these circumstances, the generated Java program runs simulator for the B+CSP specification. It generates traces, and experimentally demonstrates the properties on the traces.

4.1. Invariant Check: Simple Lift Example

Figure 5 specifies an abstract lift model. We use this simple example to demonstrate the implementation of invariant check, which demonstrates the safety properties.

Invariants in a B machine demonstrate safety properties of the system model. In the ProB model checking, the B invariants are checked on all the states of the state model. The violation of the invariants indicate an unsafe state of the system model. Implementing the invariant check in the target Java programs can provide a practical correctness demonstration for the translation strategy on safety properties.

The Java implementation of our approach supports invariants checking at runtime. The invariants supported by the translation are mainly from the B0 language conditions. The subclass of *JcspVar* class needs to implement an abstract *check* to support the checking. There are two ways to process the invariant check in the translation and implementation. The first one uses the same semantics of invariant checking as B+CSP. It forces the *check()* method to be called in every channel object after it finish its data transition. That means the invariant checking is processed at all the states of the system. However, this may seriously degrade performance in some Java applications. An alternative lightweight solution requires users to indicate the invariant check explicitly at some specific positions; the *lift* specification falls into this class. The CSP-only channel *inv_check* is used to indicate a runtime invariant check. As it has no B counterpart, it has no effect on system state. When handling this channel, the translator generates the Java code which calls the *check()* method from the subclass of *JcspVar* class. However, the alternative solution cannot guarantee all the violations of invariants being found, or discovered promptly. With a weak check, the system can run all the way through without noticing the existing violated state.

The unguarded B operations *inc* and *dec* can freely increase or decrease the B variable *floor*. That would easily break the invariant on *floor* ($level \geq 0 \ \& \ level \leq 10$). In the ProB model checking, the violated state can be quickly identified from the state model.

Runtime results of the target Java application demonstrate that the check mechanism can find violation of invariant conditions, and terminate the system accordingly. Therefore, we correct the model to that of Figure 1, by adding preconditions.

The Java programs generated from the modified specification find no violation of invariants.

4.2. Bounded Fairness Assertions

ProB also provides a mechanism to detect deadlock in the state space. When the system reaches a state where no further operation can progress, it is deadlocked. Stronger liveness properties, such as livelock-freeness and reachability, are difficult to detect in model checking, and are not supported by ProB. Fairness, which involves temporal logic, is an even more complex property for model checking. Many approaches [25,26,27] have been attempted for extending model checking of B or CSP specifications to temporal logics. However, none of these approaches can be directly supported in the B+CSP specification.

The bounded fairness assertions check is used informally to address some limited fairness properties on bounded scales. In the specification, a sequence *record* is specified. A special combined channel *rec_proc* is built to add runtime history to the *record* sequence. A CSP process can call the *rec_proc* channel with a specific ID number to record its execution. The fairness assertions are specified on a limited size of the *record* sequence.

The assertions check here is only used in Java, not ProB. Such properties cannot be model-checked in ProB because of state explosion; even an assertion with a very short window on the *record* sequence could easily explode the state space.

The translation tool and the target Java application support three kinds of bounded fairness assertions. For example:

Frequency Assertion:
$$!(i).(i \in ProcID \ \& \ card(record) > 24 \Rightarrow \mathbf{card}(\mathbf{card}(record)\text{-}24..\mathbf{card}(record) \lhd record \rhd \{i\}) > 2)$$
Duration Assertion
$$!(j).(j \in ProcID \ \& \ \mathbf{card}(record) > 12 \Rightarrow j \in \mathbf{ran}(\mathbf{card}(record)\text{-}12..\mathbf{card}(record) \lhd record))$$
Alteration Assertion
$$\mathbf{card}(record) > 3 \Rightarrow record(\mathbf{card}(record)) \notin \mathbf{ran}(\mathbf{card}(record)\text{-}3..\mathbf{card}(record)\text{-}1 \lhd record)$$

Figure 7. Bounded fairness assertions in JCSProB

The symbol !(i) here means "for all *i*", *card()* is a cardinality operator, and *ran()* returns the range of a function. The symbol $\lhd$ represents domain restriction, while the symbol $\rhd$ represents range restriction. In the example assertions, six processes are monitored. The frequency assertions try to make sure that for n (= 6) processes, in the last $4n$ record steps, each concerned process should progress more than twice. The duration assertions check the last $2n$ steps to make sure each concerned process should progress at least once. The alternation assertion check that the last progressed process does not occurs in the last three steps before that.

As our translation targets the concrete and deterministic subset of the combined specification, generally, we only support the B0 subset of B language. Many predicates and expressions in B-method are too abstract to be implemented in Java. Our bounded fairness assertions, which are defined with syntax beyond B0, are restricted to very limited formats.

In the Java application, the B sequence *record* is implemented in an array of *jcspRecord* objects. When the Java application terminates, the runtime trace is automatically saved in a log file for further investigation.

4.3. Fairness: Wot-no-chickens

The *Wot,no chickens?* example [28] was originally constructed for emphasizing possible fairness issues in the wait-notify mechanism of Java concurrent programming. There are five philosophers and one chef in this story. The chef repeatly cooks four chickens each time, puts the chicken in a canteen, and notifies the waiting philosophers. On the other hand, philoso-

phers, but not the greedy one, recursively continue the following behaviours: think, go to canteen for chickens, get a chicken, and go back to think again. The greedy philosopher doesn't think, and goes to the canteen directly and finds it devoid of chickens. The Java implementation in [28] employs the Java *wait-notify* mechanism to block the philosopher object when there are no chickens left in the canteen. The chef claims the canteen monitor lock (on which the greedy philosopher is waiting), takes some time to set out the freshly cooked chickens and, then, notifies all (any) who are waiting. During this claim period, the diligent philosophers finish their thoughts, try to claim the monitor lock and get in line. If that happens before the greedy philospher is notified, he finds himself behind all his colleagues again. By the time he claims the monitor (i.e. reaches the canteen), the shelves are bare and back he goes to waiting! The greedy philosopher never gets any chicken.

4.3.1. Two Formal Models

To test the syntax coverage of the JCSProB package and the translation, several formal models of this example are specified. We use various synchronization strategies and recursion patterns to explore the syntax coverage of the B+CSP specification in the JCSProB package, as well as in the translation tool. Furthermore, we also want to compare fairness properties of different formal models, in order to evaluate the behaviour of the generated Java programs in practice.

MACHINE *chicken*
VARIABLES
    canteen,record, ...
INVARIANT
    canteen: NAT & record: seq(NAT) .....
INITIALISATION
    canteen := 0 || record := <> ...
OPERATIONS
    .....
    getchicken(pp) =
        PRE *pp:0..4 & canteen > 0* ***THEN***
        canteen := canteen - 1 || ...
        END*;*
    .....
    put =
        BEGIN *canteen := canteen + 4 || ...* ***END***
END

———

MAIN = Chef ||| XPhil ||| PHILS ;;
PHILS = |||X:0,1,2,3@Phil(X);;
Phil(X) = thinking.X → waits.200 → getchicken.X → rec_proc.X →
    backtoseat.X → eat.X → Phil(X);;
XPhil = getchicken.4 → rec_proc.4 → backtoseat.4 → eat.4 → XPhil;;
Chef = cook → waits.200 → put → rec_proc.5 → Chef ;;

Figure 8. Formal specification of Wot-no-chicken example, Model 1

The first combined B+CSP model of this example is presented in Figure 8. The CSP part of the specification in Figure 8 only features some interleaving processes. However, the atomic access control on the B global variables, and the precondition on the *get_chicken* channel actually require synchronization mechanisms to preserve the consistency of the concurrent Java programs. As all the features concerning the concurrency model are implemented in the JCSProB package, users can work with the high-level concurrency model without noticing the low-level implementation of synchronization.

An alternative model is specified in Figure 9. As the B machine is very similar to the first one in Figure 8, only the CSP specification is given here. This model explicitly uses a multi-way synchronization on the *put* channel to force all the philosophers and the chef to synchronize.

```
MAIN = Chef [|{put}|] XPhil [|{put}|] PHILS ;;
PHILS = [|{put}|]X:{0,1,2,3}@Phil(X) ;;
Phil(X) = thinking.X → waits.200 → PhilA(X) ;;
XPhil = PhilA(4) ;;
PhilA(X) = put → PhilB(X) ;;
PhilB(X) = waits.100 → PhilA(X) [] getchicken.X → rec_proc.X →
            if(X == 4)
            then XPhil
            else Phil(X)
            ;;
Chef = waits.300 → cook → waits.200 → put → Chef ;;
```

Figure 9. Formal specification of Wot-no-chicken example, Model 2

4.3.2. Experiments and Results

The experimental evaluation test is based on the two models specified above. In the first part of the evaluation, we test the safety and deadlock-freeness properties on the two channels. In Table 4, the test results on these properties are demonstrated. The Timing column indicates how many different timing configurations are tested with the model, and the Steps column shows the lengths of the runtime records we concerned. As the concurrent Java applications constructed with the JCSProB package preserve the same safety and deadlock-freeness properties as their formal models, it partially demonstrates the correctness of the JCSProB package, as well as the translation tool.

Model Name	Property	Processes	Timing	Steps	Result
Model 1	Safety/Invariant	-	15	1000	$\surd$
Model 1	Deadlock-freeness	-	15	1000	$\surd$
Model 2	Safety/Invariant	-	15	1000	$\surd$
Model 2	Deadlock-freeness	-	15	1000	$\surd$

Table 4. The experimental result: Safety and Deadlock-freeness

To test the bounded fairness properties on the target Java programs at runtime, we first need to generate various traces from the concurrent Java programs. Currently, we use the *waits* channel in the CSP part specification to define various timing configurations for generating traces for the target Java programs. The *waits* channel forces the calling process to sleep for a fixed time period. In this way, we can explicitly animate formal models with specific timing settings for experimental purposes. Then we employ the bounded fairness assertions check on Java programs embedded with timing settings. The target of this experiment is to practically animate the Java/JCSProB applications, and evaluate their runtime performances with the bounded fairness properties.

In Table 5, we show the experimental results of the two models with bounded fairness properties. For each property, we use five different timing settings; and for each timing setting, the Java program is tested in five runs. In the result column of the table, *18P7F* means in 25 runs, the check passes 18 times and fails 7 times.

In Section 4.2, three kinds of bounded fairness assertions were introduced. In the testing, frequency and duration assertions on the formal models are checked at runtime. The asser-

Model Name	Property	Processes	Timing	Steps	Result
Model 1	Frequency 1	All	5	150	4P21F
Model 1'	Frequency 2	Phils+XPhil	5	150	1P24F
Model 1"	Frequency 3	Phils	5	150	23P2F
Model 1	Duration 1	All	5	300	20P5F
Model 1'	Duration 2	Phils+XPhil	5	300	18P7F
Model 1"	Duration 3	Phils	5	300	25P0F
Model 2	Frequency 1	All	5	150	5P20F
Model 2'	Frequency 2	Phils+XPhil	5	150	0P25F
Model 2"	Frequency 3	Phils	5	150	24P1F
Model 2	Duration 1	All	5	300	5P20F
Model 2'	Duration 2	Phils+XPhil	5	300	5P20F
Model 2"	Duration 3	Phils	5	300	25P0F

Table 5. The experimental result: Bounded Fairness Properties

tions check also concerns different process groups. In the tests on *Model 1* and *Model 2*, both the philosophers and the chef processes are recorded for assertions check. In model *Model 1'* and *Model 2'*, only the philosopher processes are run. In *Model 1"* and *Model 2"*, the greedy philosopher is removed and only normal philosopher processes are tested.

A number of points are summarized from the testing results:

- The unnecessary group synchronization in *Model 2* brings particular fairness problems to the system. The fairness properties in this model heavily depend on the timing setting. For example, all five passes for the frequency check on *Model 2* are from the same timing configuration, while the other 20 check runs on the other four different timing configurations all failed. It is mainly caused by the *wait* channel in the *PhilB(X)* process. As the greedy philosopher does not wait as other philosophers in *Phil(X)*, it enters *PhilB(X)* first and may find there is chickens there. A specific timing setting may make the greedy one waiting in *PhilB(X)*, while other philosophers take all the chickens in this time gap. In this way, we can even starve the greedy philosopher for a period of time.
- In *Model 1*, as long as the chef does not run too much faster than normal philosophers, different timings won't make the results very irregular.
- The duration assertion check also demonstrates that *Model 2* has a more serious fairness problem than *Model 1*, even with a very short trace.
- As we expect, *Model 1"* and *Model 2"*, which have no greedy philosopher, demonstrate better fairness properties than the other models.
- Further analysis of the experimental results shows that the number of channels in a process is the main factor which affects the progress of processes. For example, if we remove all the timing configurations in *Model 1*, the chef process, which has fewer combined channels than the philosopher processes, runs much faster than the five philosopher processes in the first model. The *backtoseat* and *eat* channel classes, which actually have just very simple data transitions inside the channel, result in differences in the performance. The chef keeps on producing far more chickens than the five philosophers can actually consume.

A generated Java program provides a useful simulation for its formal model. It is used to explore and discover the behaviour properties which cannot be verified in ProB model checking.

5. Conclusion and Future Work

Our implementation strategy is strongly related to a similar approach in the *Circus* development. In [29], 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 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 [30], 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.

The JCSP package (1.0-rc5) does not provide support of external choice with multi-way synchronization or output guard. As an alternative approach, we have implemented multi-way synchronization for external choice for the JCSProB channel class. As the implementation is still under test, we will report it in the future. Our plan was always to re-implement the JCSProB package with the facilities from the new JCSP package (1.0-rc7). New JCSP features, such as *AltingBarrier* and rendezvous, can be used directly to construct the implementation classes of the combined B+CSP channel. The current JCSProB implementation of combined channels has a *run()* method inside the channel class. The data transitions on system states are inside the method. That is actually very similar to a JCSP process class. The JCSProB based on JCSP 1.0-rc7 or later would see that the combined channels are implemented as special JCSP processes. They communicate with JCSP process objects, which implement CSP processes, through JCSP channels. The synchronization on the combined channel would be resolved using the *AltingBarrier* class from the new JCSP library. The data transitions would be put in the *run()* method of the process. However, although many JCSP channel classes have been formally proved in [8], the correctness proof of the *AltingBarrier* class still has still to be complete. Therefore, we still regard the re-implementation of JCSProB with new JCSP package as a future work.

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. The current JCSProB is hard to prove because it is hard to build a formal model for it. The new JCSProB channel implementation will be based on JCSP, and many JCSP channels have already been formally proved. Thus we expect that it will be modelled in CSP/*occam* and proved by FDR as before.

The other issue in the JCSProB implementation is recursion. Classical *occam* does not support recursion[2] and a `WHILE`-loop must be used for simple *tail* recursion. However, in CSP, it is very common to see a process calling other processes or itself to perform linear or non-linear recursions. In JCSP, we can employ a Java `while`-loop for any *tail* recursion in the CSP. Continually constructing and running a new process object from within the existing one to implement a never unwinding recursion must eventually cause a Java `StackOverflow` exception. To support the CSP-style recursion used in B+CSP, we implemented the existing *CSProcess* interface with a new process class. As for the multi-way synchronization classes, this recursion facility is not ready to be reported in this paper.

Considering the results of the experiments, we find that atomic access to the objects of *JcspVar* class is the most significant problem affecting the performance of the Java implementation. The exclusive lock in the subclass of *JcspVar* provides safety and consistency. We explicitly defined it because it is not only used for accessing the data, but also for our

[2] *occam*-π does support recursion.

implementation of multi-way synchronization on external choice. However, it heavily effects the performance. Applying advanced read-write techniques to replace the exclusive lock on a variable's access control may improve the concurrency performance of the Java implementation. The pragmatic solution to this problem is to provide guidance to the specifier as to how B variables may be interpreted as local CSP process variables, thus not requiring locking. For example, a member of an array of CSP processes $ProcX(i)$ might call B operation $Op(i)$, allowing B to index an array of variables, one per process. This reduces the number of global variables and thus locking load. Furthermore, future work is to implement the B+CSP channel with the new JCSP package. That means the access of the data variables and the implementation of multi-way synchronization would be separated. In this case, we could simplify the lock implementation to reduce the performance overload.

There are further outstanding issues to be resolved. We are aware that the special channels (*rec_proc*, *wait*, *inv_check*) in the invariants and assertion checking are not the best way to animate the generated Java programs and generate test cases from them. Although the channels do not affect the state of the system, this solution mixes implementation detail with the formal specification. Three solutions are under consideration:

- Configuration File. A configuration file, along with the B+CSP specification, would be used to generate the Java programs. The setting in the configuration file would guide the target Java programs to produce specific or random timing delays on the selected channels, and output system state at runtime. This can be seen as a form of specialization of the model mapping that the translation represents.
- User Interaction. A GUI interface for the target Java programs would allow users to manually manipulate the programs at runtime, producing different traces on each run.
- Traces from ProB. As an animator and model checker for the B+CSP specification, ProB can provide traces satisfying certain properties in a specific format. Using these ProB traces to guide the execution of target Java programs would be very useful.

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 composing B+CSP specification pairs, are 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 [27], an approach for composing combined B and CSP specification CSP||B is presented. Whether a similar technique is applicable for B+CSP in ProB remains to be seen.

References

[1] C.A.R Hoare, *Communicating Sequential Processes*, Prentice Hall International, 1985.
[2] R. Milner, *A Calculus of Communicating Systems*, Springer Verlag, 1980.
[3] G. Guiho and C. Hennebert, "SACEM software validation", In *Twelfth International Conference on Software Engineering*, 1990.
[4] G. T. Leavens and E. Poll and C. Clifton and Y. Cheon and C. Ruby and D. Cok and P. Müller and J. Kiniry, *JML Reference Manual*, 2005.
[5] M. Brörken and M. Möller, "Jassda Trance Assertions: Runtime Checking the Dynamic of Java Programs", In *International Conference on Testing of Communicating Systems*, 2002.
[6] J.Magee and J. Kramer, *Concurrency: State Models & Java Programs*, John Wiley and Sons, 1999.
[7] P. H. Welch and J. M. Martin, "A CSP Model for Java Multithreading", In *ICSE 2000*, pages 114-122, 2000.
[8] P.H. Welch and J.M. Martin, "Formal Analysis of Concurrent Java System" In *Communicating Process Architectures 2000*, 2000.

[9] V. Raju and L. Rong, and G. S. Stiles, "Automatic Conversion of CSP to CTJ, JCSP, and CCSP", In *Communicating Process Architectures 2003*, pages 63-81, 2003.
[10] C. Fischer, "CSP-OZ: A combination of Object-Z and CSP", Technical report, Fachbereich Informatik, University of Oldenburg, 1997.
[11] M.J. Butler, "csp2B: A practical approach to combining CSP and B", In *World Congress on Formal Methods*, pages 490-508, Springer, 1999.
[12] H. Treharne and S. Schneider, "Using a Process Algebra to Control B Operations", In *IFM 1999*, pages 437-456, 1999.
[13] J. C. P. Woodcock and A. L. C. Cavalcanti, "A concurrent language for refinement", In *IWFM01: 5th Irish Workshop in Formal Methods, BCS Electronic Workshops in Computing*, 2001.
[14] S.A. Schneider and H.E. Treharne and N. Evans, "Chunks: Component Verification in CSP||B", In *IFM 2005*, Springer, 2005.
[15] C. Fischer and H. Wehrheim, "Model-checking CSP-OZ specifications with FDR", In *IFM 1999*, pages 315-34, Springer-Verlag, 1999.
[16] M. Brörken and M. Möller, "Jassda Trance Assertions: Runtime Checking the Dynamic of Java Programs", In *International Conference on Testing of Communicating Systems*, 2002.
[17] B. Meyer, "Applying 'design by contract"', In *Computer*, volume 25, pages 40-51, 1992.
[18] M. Leuschel and M.R. Butler, "ProB: A model checker for B", In *FME 2003*, LNCS 2805, pages 855-874, Springer-Verlag, 2003.
[19] M. J. Butler and M. Leuschel, "Combining CSP and B for Specification and Property Verification", *FM 2005*: 221-236, Springer, 2005
[20] J.-R. Abrial, *The B-Book: Assigning Programs toMeanings*, Cambridge University Press, 1996.
[21] C.C. Morgan, "Of wp and CSP", In *Beauty is our business: a birthday salute to Edsger W. Dijkstra*, SpringerCVerlag, 1990.
[22] P.H. Welch, F.R.M. Barnes, and F.A.C. Polack, "Communicating Complex Systems", In I*CECCS 2006*, IEEE, 2006
[23] P.H. Welch, "A Fast Resolution of Choice Between Multiway Synchronisations", CPA-2006, IOS Press, IBSN 1-58603-671-8, 2006.
[24] P.H. Welch and Neil Brown and James Moores and Kevin Chalmers and Bernhard Sputh. "Integrating and Extending JCSP", CPA-2007, IOS Press, 2007.
[25] M. R. Hansen and E.R. Olderog and M. Schenke and M. Fränzle and B. von Karger and M. Müller-Olm and H. Rischel, "A Duration Calculus semantics for real-time reactive systems", Technical Report, Germany, 1993.
[26] M. Leuschel and T. Massart and A. Currie, "How to make FDR spin LTL model checking of CSP by refinement", In *FME'01: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, pages 99-118, Springer-Verlag, 2001.
[27] H. Treharne and S. Schneider, "Capturing timing requirements formally in AMN", Technical report, Royal Holloway, Department of Computer Science, University of London, Egham, Surrey, 1999.
[28] P.H. Welch, "Java Threads in the Light of occam/CSP", In Architectures, Laugages and Patterns for Parallel and Distributed Applications 1998, pages 259-284, IOS Press, 1998.
[29] M. Oliveira and A. Cavalcanti, "From *Circus* to JCSP", In *ICFEM 2004*, pages 320-340, 2004.
[30] A. Freitas and A. Cavalcanti, "Automatic Translation from *Circus* to Java", In *FM 2006*, pages 115-130, Springer, 2006.