# Automatic Translation from Combined *B* and CSP specification to Java Programs

Letu Yang and Michael R. Poppleton

Dependable Systems and Software Engineering
University of Southampton
Southampton, SO17 1BJ, UK
{ly03r,mrp}@ecs.soton.ac.uk

**Abstract.** A recent contribution to the formal specification and verification of concurrent systems is the integration of the state- and event-based approaches B and CSP, specifically in the ProB model checking tool. At the implementation end of the development, concurrent programming in Java remains a demanding and error-prone activity, because of the need to verify critical properties of safety and liveness as well as functional correctness. This work contributes to the automated development of concurrent Java programs from such integrated specifications.
The JCSP package was originally designed as a proven clean Java concurrency vehicle for the implementation of certain CSP specifications. In the context of best current Java concurrent programming practice, we extend the original JCSP package to support the integrated B and CSP specification by implementing new channel classes. We propose rules for the automated translation of the integrated specification to multi-threaded Java using the extended JCSP channel classes. We briefly present a prototype translation tool which extends ProB, with a worked example, and conclude with a strategy for formally verifying the translation.

## 1 Introduction

Concurrency in multithreaded Java programming has always been seen as a problematic area [Pu00], to the extent that expert practitioner advice has been to avoid it where possible [MW]. The difficulty arises from the low level of the methods provided, the responsibility of the programmer for guaranteeing various awkward concurrency properties - including safety, liveness, and fairness - and the complexities of scale. The recent JDK 5.0 issue [Go04] has improved matters somewhat by raising the level of abstraction in the concurrency model, introducing constructs such as *semaphore* and *mutex*. Abstraction has also been raised, principally in package *util.concurrency* by deprecating low-level *Thread* methods such as *stop, resume* and *suspend* and replacing them with high-level thread-safe facilities. Safety properties have been made more tractable by the provision of a common cross-platform Java Memory Model [MPA05]. However, as the concurrency model of Java programs is described in natural language, it is still difficult to detect and avoid liveness and fairness problems in such programs.

The difficulty of concurrency motivated the development of formal languages for modelling concurrent processes such as CSP [SS00] and CCS [RM89]. The capability of formal and automated [MK99,FS03] verification of safety and liveness properties in such concurrency models, before transformation into code, has added real value to industrial systems, including hardware systems[JP04], software systems[JL04], and communication protocols [SD04]. Formal analysis techniques have been applied to concurrent Java programs: [LP05] and [BM02] provide languages to add assertions to Java programs, and employ runtime verification techniques to verify 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, represents a useful contribution.

One recent trend in formal approaches to system design is to integrate the state- and event-based approaches. 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 modelling capabilities, in both aspects. [Bu99,TS99] have proposed the integration of the state-based B method [Ab96] and CSP, an event-based process algebra. The ProB [BL05] tool supports model checking of combined B and CSP specifications[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.

Peter Welch's JCSP package [PM00a] provides a high-level concurrency model for Java, implementing the *Occam* language [ST95], a concurrent programming language that directly implements a subset of CSP. JCSP is based on the point-to-point communication model of *Occam*. The correctness of the JCSP translation of the *Occam* channel to a JCSP channel class has been formally proved [PM00b]: the CSP model of the JCSP channel communication is shown to refine the CSP/*Occam* concurrency model. Raju [RR03] has developed a tool to translate subset CSP models directly to JCSP. The tool does not extend beyond the *Occam* subset of CSP, and does not scale, in our experience, beyond small textbook examples. Furthermore, through our experience with JCSP and Raju's tool, we find that the point-to-point *Occam* communication model limits the capability of JCSP for developing concurrent systems based on other concurrency models.

Being event-based, CSP is insufficiently expressive of the data aspect of systems; JCSP is similarly limited. In [RR03], data declaration and assignment are manually added to the Java programs generated by the tool; this can easily break the correctness of the system model which is proved by FDR tool.

Motivated by both the Java concurrency issues and the integrated formal method approaches, we present a translation strategy, which converts combined B and CSP specifications in ProB into Java programs using an extended JCSP

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

package. The design of the translation rules has taken some inspiration from [OC04], which defines a translation from *Circus* to JCSP.

In Section 2, we briefly introduce the existing JCSP package. We then discuss the reasons for extending the original package, and the new features of the extended JCSP package. In Section 3, we demonstrate the translation strategy from the combined B and CSP specification to Java programs, and discuss the translation tool which implements these rules in Prolog. In Section 4, we illustrate the translation with a Chef-and-Dining-Philosophers example. In Section 5, we outline the verification required of this translation process as future work. Section 6 discuss future work and conclusions.

## 2 The Extended JCSP package

### 2.1 JCSP

JCSP [PM00a] is a Java package for developing concurrent Java Programs. It implements the *Occam* subset of the CSP language [SS00], as well as some other features of CSP, in a series of process and channel classes. The *Occam* language definition is based on that of CSP, but is aimed at modelling channel communication. *Occam* channels are based on CSP events. However, *Occam* channels are only applied for modelling point to point communication, while CSP events are used in more general concurrency models. In Figure 1, the concurrency model of JCSP and *Occam* is briefly illustrated in CSP syntax. CSP processes *ProcA* and *ProcB* synchronize with each other on channel $A$, while B and C are unsynchronized channels. The synchronization happens when *ProcB* is ready to output data $y$ at $A!y$, and *ProcA* is ready to input data $x$ at $A?x$.

$$ProcA = A?x \rightarrow B \rightarrow ProcA$$
$$ProcB = C \rightarrow A!y \rightarrow ProcB$$

**Fig. 1.** A CSP specification for Occam/JCSP concurrency

A Java application using the $JCSP$ package consists of a number of objects of $JCSP$ process classes running in parallel. The process objects communicate with each other via objects of JCSP channel classes. All the process classes used here implement an abstract JCSP process class `CSProcess`. The channel classes all inherit the `inputchannel` and `outputchannel` interfaces of the JCSP package. JCSP supports two process objects synchronizing and communicating data through a JCSP channel object. A JCSP process blocks when it requires a data communication with the other process on a specific channel. Only when both the output and input side processes of the channel are ready for the data communication, is the data is transmitted through the channel.

Since the JCSP channel classes implement only the *Occam* channels (as opposed to more general CSP events) the communication between two processes is the only synchronization supported by JCSP channel classes. Although

*Any2OneChannel* and *Any2AnyChannel* classes handle more than two processes, the synchronization still only involves two processes. For *Any2OneChannel*, many writer processes and one reader process are associated on the channel. All the requests from writer processes are grouped into a queue. At a given time, only the reader and one writer actually synchronize with each other and pass data through the channel. Thus the synchronization model still is the point-to-point communication of *Occam*.

JCSP does support synchronization between more than two channels with a *Barrier* class. However, *Barrier* is not implemented as a JCSP channel class, and uses a simple counter to resolve the synchronization. Therefore it can not help to do multi-way synchronization in a manner faithful to the CSP concurrency model.

### 2.2 Why We Extend JCSP

As the JCSP package was designed to implement the *Occam* subset of CSP in Java, it cannot be directly used for translating the combined B and CSP specification in ProB. The reasons for this are twofold.

For the CSP part, the combined specification uses a bigger CSP subset than that of JCSP package. Important CSP language features, especially some concurrency facilities such as Alphabetized Parallel, are not supported by JCSP. In developing a concurrent Java system, the synchronization between more than two threads on a certain data transition is a typical concurrency problem, which can be easily specified using CSP. This problem is also identified by JDK5.0, and in *java.util.concurrency* package, a *CyclicBarrier* class is build to implement this synchronization in a high-level facility. However, modelling this kind of concurrency in *Occam* gives the programmer extra work implementing the synchronization on the shared thread. Implementing them in JCSP also requires extra facility classes to resolve the synchronization.

Figure 2 shows the CSP specification of a parallel hardware interface. The interface reads eight bits from eight different channels, and when all the bits are ready, it generates a byte from the bits. In the specification, each bit process *B(i)* gets a bit using event *get(i)?bit*, and then waits for the *makebyte* event. The bit processes interleave with each other. The parallel interface process *PI* repeatedly makes bytes with the *makebyte* event, when all the bits are ready. The *Main* process requires all the processes to synchronize on the *makebyte* event. As the CSP model of this example has nine processes synchronizing on the same event, it can not be directly translated into Java using the original JCSP package.

The channel used in JCSP and *Occam* is obliged to communicate data between two processes, while the combined channel we propose uses a more general definition of CSP event. The CSP event can optionally have data parameters, with decorations denoting input, output or dot data, which can be either input or output. Parameters are similarly specified in the B+CSP specification. In the extended JCSP, we implement these kind of CSP events in Java with the new channel class.

$B(i) = get(i)?bit \rightarrow makebyte.byte \rightarrow B(i)$
$PI = makebyte.byte \rightarrow PI$
$Bsys = ||| \ i{:}1..8@ \ B(i)$
$Main = PI \ [|\{makebyte\}|] \ Bsys$

---

* $|||$ *declares a set of interleaving processes*
* $A[|c|]B$ *is the syntax for Alphabetized Parallel, which means that processes A and B synchronize on a set of events c*

**Fig. 2.** CSP specification of Parallel Interface

Since the JCSP package is designed for implementing CSP specifications in Java, it has no facility to implement the B part of the combined specification in Java. Therefore we need a strategy for translating the B specifications into Java. Fortunately, the B specification used for the automatic translation is mainly from the B0 subset of the B language, which represents a simple programming language designed to be automatically translatable to target languages of choice. The B+CSP channel combines a B operation wih a CSP event; the operational semantics of this is given in [BL05].

### 2.3 The Extended JCSP Package

The JCSP package is extended with new channel and facility classes. The new "parallel/choice" channel class `PaChoChannel` implements the features of combined B and CSP specifications which are not included in the original JCSP package. The facility classes implement external choice for the extended channel class.

The extended channel and facility classes are designed as an add-on package for the original JCSP. JCSP process classes can declare and use the new channel classes in a similar manner to using the original JCSP channel classes. As all the changes have been preserved inside the channel and facility classes, there are no significant changes in using them in process classes.

Class `PaChoChannel` supports synchronization between more than two processes. It keeps track of all the processes associated with this channel. When all the associated processes are ready, the data operations in the channel are triggered. After the data operations complete, all the associated processes are notified.

`PaChoChannel` also supports the *dot* event $c.v$, where $c$ is a CSP event and $v$ is a data item on it. The synchronization of *dot* data channels is not only decided by the name of the channels, but also by the *dot* data values. Two processes, using the same channel $c$ but with different values of $v$ (e.g. $c.x$, $c.y$), will not synchronize with each other. This implementation is based on the B+CSP semantics. The input $c?x$ and output $c!x$ events are also supported by the new channel class.

With the implementation of the above two CSP language features, the new JCSP package can support *Alphabetized Parallel* of CSP, an important facility for

specifying concurrent systems. Furthermore, some new facility classes implement *external choice* for the extended channel classes.

The main issue for implementing the B part of the combined specification is how to implement the data transitions of a B operation into a combined channel class. The *PaChoChannel* class implements the *Serializable* interface of Java, and has an abstract method *run*. The subclasses of *PaChoChannel* overwrite the *run* method by putting in the target Java code of the B data transitions. The *run* method is executed when all the associated processes are ready for the execution of the channel class.

Thus, the extended JCSP package supports a bigger subset of CSP than the original JCSP, as well as providing facilities to implement the B part of the combined specifications in Java. This makes it possible to translate the combined B and CSP specifications into Java programs. An example with the extended channel class is discussed in Chapter 4.

The synchronization supported by the extended channel class is implemented with concurrency primitives from Java monitors, which exclude the facilities deprecated by Java 5.0. The correctness of this implementation needs to be verified; a formal proof strategy is proposed later in Section 5.

## 3 Translation Strategy and Tool

A series of translation rules are developed to structure the automatic translation. The rules are used recursively to generate a set of Java classes from a combined B and CSP specification. We discuss some of the key translation rules in this section. The translation tool implements the translation rules in Prolog.

### 3.1 Translation Strategy and Rules

In the translation, each combined B+CSP channel is translated into an object of a channel class. Each process in the CSP part of the combined specification is translated into an object of a JCSP process class. Indexed processes are translated into different objects of the same JCSP process class. Their indices are treated as parameters of the JCSP class constructor.

**Translation of the MAIN Process** The translation strategy is mainly based on the execution behaviour of the system which is specified in the `MAIN` process, which is the core process of the CSP part. The translation rules set generates the executable Java class for the target application. It starts with the MAIN process with rule *Main Proc Decl*, and recursively generates process classes for all the associated CSP processes.

Through this procedure, the translation gathers information of all the CSP channels $v$, B sets $S$, and variables and constants $s$. Rule *Set Def* generates Java classes to represent B set $S$. All the variables are declared and initialized by rule *Par Def*. The rule generates the declaration and initialization of the variables with the information from the B part. $v$ is expanded with rule *Ch Def*, which will

declare the channel objects for all the channels, and generate channel classes. Finally, it generates the code for declaring the MAIN process object, including the code of the ***run*** call.

| Name | *Main Proc Decl* |
|---|---|
| CSP | **MAIN** |
| B | **MACHINE** M |
| Java | $\langle MAIN \rangle^{ProcDecl}$<br>**public class** M_**machine {**<br>**public static void main(){**<br>$\langle S \rangle^{SetDef}$<br>$\langle s \rangle^{ParDef}$<br>$\langle \upsilon \rangle^{ChDef}$<br>**new** $\langle M \rangle^{ProcCName}$**(**$\langle s \rangle^{ParList}$,$\langle \upsilon \rangle^{ChList}$**).run();**<br>**}**<br>**}** |

**Table 1.** Rule 0: Rule for Declaring Main Process

**Support of Multi-way Synchronization** One important feature of the extended JCSP channel is the support of multi-way synchronization. The parallel structure is handled by rules from the rule set *ProcE*. For example, the combination of the indexed parallel processes is translated by rule *ProcE (Re-Parallel)*.

| Name | *ProcE (Re-Parallel)* |
|---|---|
| CSP | [\| $\upsilon$ \|]n:a@ P(n) |
| Java | **CSProcess[] procs = {**<br>**new** $\langle P \rangle^{ProcCName}$**(**$\langle s \rangle^{ParList}$,$\langle \upsilon \rangle^{ChList}$,$\langle \upsilon_1 \rangle^{ChList}$,$a_1$**),**<br>.....<br>**new** $\langle P \rangle^{ProcCName}$**(**$\langle s \rangle^{ParList}$,$\langle \upsilon \rangle^{ChList}$,$\langle \upsilon_n \rangle^{ChList}$,$a_n$**)**<br>**}**<br>**new Parallel(procs).run();** |

**Table 2.** Rule 9: Rule for Replicated Alphabetized Parallel

In rule ProcE (Re-Parallel) (Table 2), $\langle P \rangle^{ProcCName}$ refers to the name of the indexed process class, while each process object from $P(a_1)$ to $P(a_n)$ is an instance of that process class. These process objects synchronize on a set of channels $\upsilon$. Each process object may include a set of channels $\upsilon_n$, which do not synchronize with other indexed processes here. ***PaChoChannel*** class provides ***ready*** methods to support multi-way synchronization. When a JCSP process is ready for the execution of a channel, it calls the ***ready*** method of the channel, and waits for other processes which also synchronize on this channel. As there may or may not be data on the channel, different implementations of the ***ready*** methods are provided. Table 3 shows the different rules for translating the ***ready*** call.

Ready call rules for input/no input on channel are given in Table 3. The ready calls including output parameters are discussed in the following section.

For rule Par Vec, the set $is$ of input parameters are grouped into `Vector`, and performs as a single parameter for `ready` method.

| | |
|---|---|
| Name | *Ch Call (ready I)* |
| CSP | *cc.is* |
| B | *cc(is) Instruction* |
| Java | $\langle$cc$\rangle^{ChName}$**.ready(**$\langle is \rangle^{ParVec}$**);** |
| Name | *Ch Call (ready II)* |
| CSP | *cc* |
| B | *cc Instruction* |
| Java | $\langle$cc$\rangle^{ChName}$**.ready();** |

**Table 3.** Rule 6: Rule for Channel Call , ready

**Translation of Combined Channels** The main issue in translating the combined B+CSP channels is how to resolve the parameters from both B and CSP sides. In the original JCSP, as there is no data transition inside the channel, a data $x$ on a channel $Ch$ is simply passed through the channel. Here a process with channel $Ch!x$ synchronizes with a process with $Ch?x$ event.

| | |
|---|---|
| Name | *ChC* |
| B | *os* $\leftarrow$ *cc(is) Instruction* |
| Java | **public class** $\langle$ *cc* $\rangle^{ChCName}$ **extends PaChoChannel {**<br>$\langle$ *is* $\rangle^{ParDef}$<br>$\langle$ *os* $\rangle^{ParDef}$<br>**public** $\langle$ *cc* $\rangle^{ChCName}$(**(**$\langle$ *is* $\rangle^{ParDef}$**){**<br>$\langle$ *is* $\rangle^{ParRel}$<br>**}**<br>**public void run(){**<br>$\langle$ *Instruction* $\rangle^{BInstruction}$<br>**}**<br>**}** |

**Table 4.** Rule 29: Rule for Extended Channel Classes

Therefore, in the target Java program, two process classes use the `read` and `write` methods of JCSP channel classes to communicate the data:

```
Process 1:
  ...
  run(){
      ...; Ch.write(x); ...
  }
...
```

```
Process 2:
  ...
  run(){
      ...; x = Ch.read(); ...
  }
...
```

To resolve the parameter issue for the combined B+CSP channel, we need to examine the operational semantics of B+CSP in [BL05]. The operational

semantics of the combined B+CSP channel are: $(\sigma,P) \rightarrow_A (\sigma',P')$. $\sigma$ and $\sigma'$ are the before and after B states for executing operation $o$, while $P$ and $P'$ are the before and after processes for processing channel $ch$. The combined channel A is a unification of the CSP channel $ch.a_1, ..., a_j$ and the B operation $o = o_1,...,o_m \leftarrow \text{op}(i_1,...,i_n)$, where $j = m + n$. Therefore, the combined B+CSP channel A can be expressed as: $A.s_1.....s_m.s_{m+1}.....s_{m+n}$.

In the translation, both $o_1$ ... $o_m$ and $i_1$ ... $i_n$ are treated as parameter lists for the channel classes. In rule *ChC* (Table 4), $o_1, .., o_m$ are translated into Java objects $os$, which are the output parameters, and $i_1, ..., i_n$ are translated into $is$, which are the input parameters. Rule *Par Def* obtains information from the B specification as before. In the channel class, $is$ are static parameters whose values won't be changed. $os$ are private parameters of the channel class and are made externally visible by being returned by the `ready` method. $os$ are returned as a Java `Vector`, as defined in rule *Par Vec*.

The two translation rules in Table 5 show how the process gets the output parameters from the `ready` call.

| Name | *Ch Call (ready III)* |
|---|---|
| CSP | *cc.is.os* |
| B | os ← *cc(is) Instruction* |
| Java | $\langle os \rangle^{ParVec} = \langle \text{cc} \rangle^{ChName}$**.ready(**$\langle is \rangle^{ParVec}$**);** |
| Name | *Ch Call (ready IV)* |
| CSP | *cc.os* |
| B | *os* ← *cc Instruction* |
| Java | $\langle os \rangle^{ParVec} = \langle \text{cc} \rangle^{ChName}$**.ready();** |

**Table 5.** Rule 6 (continue): Rule for Channel Call , ready

**Translation of B** B0 is a concrete low-level deterministic imperative programming language. It is the target language for generating concrete programs from verified abstract B machines, and it is translatable to high-level programming languages [BB1,VT02].

The B0 language only includes concrete B substitutions and only handles concrete data. It is easy to correctly find corresponding data instructions in high-level programming languages for the concrete substitutions. There are two correctness issues in translating B0 into a high-level programming language. The first is how to translate parameter passing of B operations to the high-level programming language. The above discussion on translating the combined B+CSP channel answered this issue for our translation. The other issue concerning the correctness of the translation is how to represent some B0 data structures in a high-level language. Usually, high-level languages provide better support for the arithmetic data, such as integer and boolean. The only concern here is some complex data structures, such as sets and arrays. Java fully supports array structures which easily represent the B0 array. Currently B0 sets are translated into *enum* static classes.

### 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

In this section we illustrate how the translation tool works. The example we used here is a simplified version of the *Wot, no Chicken* example from [We00]. The example was originally constructed for emphasizing fairness issues in the wait/notify strategy of Java concurrent programming. We use the example to demonstrate the automatic production of a concurrent Java program from a B+CSP specification. Our version of the example is simplified in omitting the lazy philosopher who raises the fairness issues.

This example includes a chef who cooks chicken, a canteen which is used to store the chicken, and several philosophers who consume the chicken. The chef spends some time to cook a number of chickens and then put them in the canteen. The philosophers take time to think, then take chicken from the canteen and eat. The CSP specification in Figure 3 shows the behaviour of the system. The *Main* process is the core process. It consists of a *Chef* process, and several *Chicken(i)* processes and Philosopher processes *Phil(i)*. The *Phil(i)* processes do not synchronize with each other, while all the *Chicken(i)* processes synchronize on the *put* event. The *Phil(i)* processes and *Chicken(i)* processes synchronize on the *getchicken(i)* event. The *Chef* process synchronizes with other processes on the *put* event.

$$
\begin{aligned}
&Main = SYSTEM[|\{put\}|]Chef\\
&Phil(i) = gotocanteen.i \rightarrow getchicken.i \rightarrow backtoseat.i \rightarrow eat.i \rightarrow thinking.i \rightarrow Phil(i)\\
&Chef = cook \rightarrow put \rightarrow Chef\\
&Chicken(i) = put \rightarrow getchicken.i \rightarrow Chicken(i)\\
&PHILS = ||| \ i{:}N@\ Phil(i)\\
&CHICKENS = [|put|]\ i{:}N@\ CHICKEN(i).\\
&SYSTEM = PHILS[|\{getchicken\}|]CHICKENS
\end{aligned}
$$

**Fig. 3.** Chef-Philosophers example: CSP part

The B specification in Figure 4 gives a part of the B specification of the combined specification. It shows the canteen and philosopher part of the chef-philosopher example. All the B operations use SELECT statements instead of PRE statements. PRE aborts when the condition is not satisfied, which means it is not guaranteed to terminate.

```
MACHINE chicken
......
SETS
    PhilStates = {thinking, hungry, full} ;
......
OPERATIONS
    ......
    gotocanteen(pp) =
        SELECT pp:Phils THEN
        state(pp) := hungry
    END;
    getchicken(pp) =
        SELECT pp:Phils THEN
        chicken(pp) := 1 ||
        canteen := canteen - 1
    END;
    eat(pp) =
        SELECT pp:Phils THEN
        chicken(pp) := 0 ||
        state(pp) := full
    END;
    backtoseat(pp) =
        SELECT pp:Phils THEN
        state(pp) := eating
    END;
    ......
END
```

**Fig. 4.** Chef-Philosophers example: B part

The execution of the B operations are guarded by the CSP specification. Therefore, the translation tool generates a JCSP process object for each process in the CSP specification.

In Figure 5, the `Phils` process groups all the interleaving `Phil` processes into a process array. Using the translation rule for indexed interleaving, which is similar to the rule for indexed parallel in Table 2, the target *Phils.java* class builds an array `procs` for all the `Phil` process objects, and runs all of them in parallel. All the associated channel objects are passed to the process object through its constructor as parameters. The index numbers of all the `Phil` processes are also passed to them as parameters. A target JCSP process class, which is translated

```
public class Phils implements CSProcess{
  PaChoChannel gotocanteen;
  PaChoChannel getchicken;
  PaChoChannel backtoseat;
  PaChoChannel eat;
  PaChoChannel thinking;
  /* Constructor of the class */
  public void run(){
    CSProcess[] procs = {
      new Phil(gotocanteen, getchicken,
        backtoseat, eat, thinking, 1),
      new Phil(gotocanteen, getchicken,
        backtoseat, eat, thinking, 2),
      new Phil(gotocanteen, getchicken,
        backtoseat, eat, thinking, 3),
      new Phil(gotocanteen, getchicken,
        backtoseat, eat, thinking, 4),
      new Phil(gotocanteen, getchicken,
        backtoseat, eat, thinking, 5)
    };
    new Parallel(procs).run();
  }
}
```

**Fig. 5.** Target Java Class: *Phils.java*

from the `Phil(i)` process, is shown in Figure 6. All the channel objects in the *Phil.java* are created by the superior process `Phils`. So a process `Phil(i)` needs to synchronize on some shared channel objects with other processes. The `Phil` process objects synchronize with `Chicken` process objects on *getchicken.n* chan-

```
public class Phil implements CSProcess{
  PaChoChannel gotocanteen;
  PaChoChannel getchicken;
  PaChoChannel backtoseat;
  PaChoChannel eat;
  PaChoChannel thinking;
  Integer num;
  /* Constructor of the class */
  public void run(){
    while(true){
      gotocanteen.dotready(this,num);
      getchicken.dotready(this,num);
      backtoseat.dotready(this,num);
      eat.dotready(this,num);
      thinking.dotready(this,num);
    }
  }
}
```

**Fig. 6.** Target Java Class: *Phil.java*

nel objects. The execution sequence of all channel objects in the `run` method implements the trace semantics of the CSP process `Phil(i)`. When the process is ready for the execution of a channel object, it calls the `ready` method, blocks itself, and waits for other processes, which also synchronize on this channel, to be ready for execution.

In the process class, channel objects are declared as instances of `PaChoChannel` classes. Actually, they have their own channel classes which extend `PaChoChannel` class. Therefore, when a undefined channel class is referred, it needs to be generated from the combined specification of the channel. The *Eat.java* class in Figure 7 is the target channel class of `eat` channel. This channel class is generated using translation rule ChC, which is discussed in the previous section.

```
public class eat extends PaChoChannel{
  Integer[] chicken;
  PhilStates state;
  public getchicken(Integer[] chicken,
       PhilStates[] state){
    super();
    this.state = state;
    this.chicken = chicken;
  }
  public synchronized void run(){
    Integer dotvalueint =
      (Integer)curdotvalue;
    chicken[dotvalueint.intValue()] = 0;
    state[dotvalueint.intValue()] =
      PhilStates.FULL;
  }
}
```

**Fig. 7.** Target Java Class: *Eat.java*

In the `run` method of the `eat` class, `chicken` is a global array which records the number of chickens that each philosopher has. Changing the `chicken` record of the current philosopher to 0 implements the data transition `chicken(pp) := 0` in the B operations. `PhilStates` is a enumeration class which indicates the status of a philosopher. It can be THINKING, HUNGRY and FULL. After a philosopher eats a chicken, his status changes to FULL. The global array `state` is used to store the status of all the philosophers. Changing the status of the current philosopher to FULL implements the data transition `state(pp) := full` in B operation `eat`.

## 5 Correctness Proof Strategy

A correctness verification is required for the translation. In [RR03,OC04], the translations are discussed without considering the correctness of the translations. Formal verification which proves the correctness of the translation in terms of semantic models of the specification and Java programs respectively would be the best solution. We propose a more modest approach based on [WM00] for future work. The new *PaChoChannel* class is designed to represent the behaviour of the combined B+CSP channel in the target Java application. To use the new channel class more confidently, we still need to formally prove that it is a correct implementation of the combined B+CSP channel.

In [WM00], the correctness of Welch's original JCSP channel classes is proved. Each JCSP channel class (i.e. Java implementation) is formally specified by a CSP model. The desired channel behaviour (which the JCSP class implements) is also specified in CSP. The refinement checking tool FDR is used to prove the two CSP models equivalent: that is, JCSP refines CSP and CSP refines JCSP. A number of such proofs are required: there are a number of JCSP channel classes, implementing the various capabilities of CSP channels. The proof strategy starts with the simple *One2OneChannel* class without alternation, and gradually builds formal models for more complex JCSP channel classes.

To prove the correctness of the *PaChoChannel* class, a similar strategy is proposed: to prove that the implementation refines the specification. First, a B+CSP model for the *PaChoChannel* class is built. Then, the required behaviour of the combined B+CSP channel is specified with B+CSP specifications. As the ProB tool supports refinement checking between B+CSP models, we can prove the *PaChoChannel* class correctly implements (i.e. refines) the B+CSP channel.

The construction of a concrete model for the *PaChoChannel* class with full functionality will be a significant task, as will its verification. Hence we would start with an abstract model of *PaChoChannel* with simple functionality, gradually building concrete models with incremental data and concurrency capabilities.

## 6 Related Work and Conclusion

Our work aims at automatically generating concurrent Java programs from proven formal specifications. To achieve this, we extend the original JCSP package to implement the combined B and CSP specification in Java. A set of translation rules are developed to formalize the translation, and the automatic translation tool is built upon the translation rule set. We also propose a formal verification strategy for proving the correctness of the translation.

There is a similar tool [RR03] to translate a pure CSP specification into a Java program using the original JCSP package. As it is not very convenient to use CSP specification to model data aspect of systems, the target Java code of this tool always needs further manual revision to add data elements. However, manual revision has the danger that the concurrency model of JCSP may be

broken by such revision. From our experience, the tool only works on some specific examples, and seems unstable.

In [OC04], a set of translation rules are proposed for translating a subset of the *Circus* specification language to Java using JCSP. The translation is restricted because the JCSP package lacks the ability to implement data transitions of the *Circus* language. Therefore, [OC04] proposes future work to extend JCSP to support the full *Circus* semantics. They also plan to develop an automatic translation tool using their rules.

[MK99] presents a strategy of using a process algebra language, $FSP$ (Finite State Processes) to build a formal concurrency model of Java concurrent programming. The $LTSA$ (*Labelled Transition System Analyser*) tool is adopted to translate the $FSP$ descriptions to a graphical representation. It also checks desirable and undesirable properties of the $FSP$ model. However, it doesn't provide exhaustive rules or tool support to link the $FSP$ syntax with the Java language. The development of the concurrent Java code relies on users' experience of this approach. Correctness of the target Java program cannot be proved formally.

JML [LP05] and Jassda [BM02] are runtime verification approaches for using formal methods to help develop concurrent Java programs. They both have assertion languages to specify pre- and post-conditions, and temporal properties of Java programs. The assertions can be checked at runtime to see whether they are preserved during the execution of the Java programs. The Java programs still need to be constructed manually.

An ambitious project [VH00] developed a tool *Java Path Finder* (JPF), which integrates model checking, program analysis and testing for Java programs. The JPF tool can generate a state model of the Java program via the support of its own Java Virtual Machine($JVM^{JPF}$). Accordingly, formally defined properties and assertions can be verified in the state model. To avoid state explosion, the Java language features that can be used in $JPF$ are restricted.

Future plans include a substantial case study using the translation tool. The stability and scalability of the translation strategy and the tool will be the focus of this exercise. The development of a GUI (Graphical User Interface) is also planned. It will provide facilities for configuring the translation, and interfacing with the target Java application.

**Acknowledgements:** We would like to thank Denis A. Nicole for his very helpful comments.

## References

[BB1] Didier Bert, Sylvain Boulmé, Marie-Laure Potet, Antoine Requet, Laurent Voisin.: Adaptable Translator of B Specifications to Embedded C Programs In FME 2003: Formal Methods, LNCS 2805, pp. 94-113, Pise, Sept. 2003.

[BL05] M. J. Butler and M. Leuschel.: Combining CSP and B for Specification and Property Verification. FM 2005: 221-236

[BM02] M. Brörken and M. Möller.: Jassda Trance Assertions: Runtime Checking the Dynamic of Java Programs. In International Conference on Testing of Communicating Systems, 2002.

[Bu99] M. J. Butler.: csp2B: A Practical Approach to Combining CSP and B. In World Congress on Formal Methods, pages 490-508, 1999.
[FS03] Formal Systems(Europoe) Ltd.: Failures-Divergence Refinement: FDR2 User Manual, 2003
[Go04] B. Goetz.: Concurrency in JDK 5.0. Technical report, IBM, 2004.
[JL04] J. Lawrence.: Practical Application of CSP and FDR to Software Design. In 25 Years Communicating Sequential Processes, 151-174, 2004.
[JP04] J. Peleska.: Applied Formal Methods - From CSP to Executable Hybrid Specifications. In 25 Years Communicating Sequential Processes, 293-320, 2004.
[Ab96] J.-R. Abrial.: The B-Book: Assigning Programs toMeanings. Cambridge University Press, 1996.
[LP05] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Muller, and J. Kiniry.: JML Reference Manual. 2005.
[MK99] J.Magee and J. Kramer.: Concurrency: State Models & Java Programs. John Wiley and Sons, 1999.
[MPA05] J. Manson, W. Pugh, and S. V. Adve.: The Java Memory Model. In POPL 05: Proceedings of the 32nd ACM SIGPLAN-SIGACT , 378-391, New York, NY, USA, 2005. ACM Press
[MW] H. Muller and K. Walrath.: Threads and Swing. http://java.sun.com/products/jfc/tsc/articles/threads/threads1.html
[OC04] M. Oliveira and A. Cavalcanti.: From Circus to JCSP. In Sixth International Conference on Formal Engineering Methods, November 2004.
[Pu00] W. Pugh.: The Java Memory Model is Fatally Flawed. Concurrency: Practice and Experience, **12(6)**(2000) 445-455
[PM00a] P. H. Welch and J. M. Martin.: A CSP Model for Java Multithreading. In ICSE 2000, pages 114-122, June 2000.
[PM00b] P. H. Welch and J. M. Martin.: Formal Analysis of Concurrent Java System. In Communicating Process Architectures 2000.
[RM89] R.Milner.: Communication and Concurrency. Prentice-Hall, Inc., 1989.
[RR03] V. Raju, L. Rong, and G. S. Stiles.: Automatic Conversion of CSP to CTJ, JCSP, and CCSP. In Communicating Process Architectures 2003, pages 63-81, 2003.
[SD04] S. Schneider and R. Delicata.: Verifying Security Protocols: An Application of CSP. In 25 Years Communicating Sequential Processes, 243-263, 2004.
[SS00] S. Schneider.: Concurrent and Real-Time System: The CSP Approach. John Wiley and Sons LTD, 2000.
[ST95] S.-T. M. Limited.: Occam 2.1 Reference Manual, 1995.
[TS99] H. Treharne and S. Schneider.: Using a Process Algebra to Control B Operations. In IFM, pages 437-456, 1999.
[VH00] W. Visser, K. Havelund, G. Brat, and S. Park.: Model checking programs. In Int. Conf. on Automated Software Engineering, 2000
[VT02] J. C. Voisinet, B. Tatibouet and A. Hammand.: JBTools: An experimental platform for the formal B method. In PPPJ' 2002 , 137-140, 2002.
[We00] P. Welch. Wow, no chicken? http://wotug.ukc.ac.uk/parallel/groups /wotug/java/discussion/3.html.
[WM00] P. H. Welch and J. M. Martin.: Formal Analysis of Concurrent Java System. In Communicating Process Architectures, 2000