From *Circus* to JCSP

Marcel Oliveira and Ana Cavalcanti

Department of Computer Science – The University of York
York, YO10 5DD, England

Abstract. *Circus* is a combination of Z, CSP, and Morgan's refinement calculus; it has an associated refinement strategy that supports the development of reactive programs. In this work, we present rules to translate *Circus* programs to Java programs that use JCSP, a library that implements CSP constructs. These rules can be used as a complement to the *Circus* algebraic refinement technique, or as a guideline for implementation. They are a link between the results on refinement in the context of *Circus* and a practical programming language in current use. The rules can also be used as the basis for a tool that mechanises the translation.

Keywords: concurrency, object-orientation, program development.

1 Introduction

Languages like Z, VDM, Abstract State Machines, and B, use a model-based approach to specification, based on mathematical objects from set theory. Although possible, modelling behavioural aspects such as choice, sequence, parallelism, and others, using these languages, is difficult and needs to be done in an implicit fashion. On the other hand, process algebras like CSP and CCS provide constructs that can be used to describe the behaviour of the system. However, they do not support a concise and elegant way to describe complex data aspects.

Many attempts to join these two kinds of formalism have been made. Combinations of Z with CCS [4, 17], Z with CSP [15], and Object-Z with CSP [2] are some examples. Our work is based on *Circus* [19], which combines Z [20] and CSP [6, 14] and, distinctively, includes refinement calculi constructs and provides support for refinement in a calculational style, as that presented in [9].

Circus characterises systems as processes, which group constructs to describe data and behaviour. Z is used to define the data aspects, and CSP, Z schemas, and guarded commands are used to define behaviour. The semantics of *Circus* is based on the unifying theories of programming [7], a relational framework that unifies the programming theory across many different computational paradigms.

The main objective of this work is to provide a strategy for implementing *Circus* programs in Java. The strategy is based on a number of translation rules, which, if applied exhaustively, transforms a *Circus* program into a Java program that uses the JCSP [13] library. These rules capture and generalise the approach that we took in the implementation of a large case-study in *Circus*.

The result of refining a *Circus* specification is a program written in a combination of CSP and guarded commands. In order to implement this program, we

need a link between *Circus* and a practical programming language. The transformation rules presented in this paper create this link. The existence of tool support for refinement and automated translation to Java makes formal development based on *Circus* relevant in practice. Our rules can be used as a basis in the implementation of a translation tool.

We assume that, before applying the translation strategy, the specification of the system we want to implement has been already refined, using the *Circus* refinement strategy presented in [**?**]. The translation strategy is applicable to programs written in the executable subset of *Circus*.

In Section 2, we use an example to introduce the main constructs of *Circus*. Section 3 presents JCSP with some examples. The strategy to implement *Circus* programs using JCSP is presented in Section 4. Finally, in Section 5 we conclude with some considerations about the strategy, and describe some future work.

2 *Circus*

Circus programs are formed by a sequence of paragraphs, which can either be a Z paragraph, a declaration of channels, a channel set declaration, or a process declaration. In Figure 1, the syntactic categories N, Exp, Pred, SchemaExp, Par, and Decl are those of valid Z identifiers, expressions, predicates, Z schemas, paragraphs in general, and declarations, respectively, as defined in [16].

We illustrate the main constructs of *Circus* using the specification of a simple register (Figure 2). It is initialised with zero, and can store or add a given value to its current value. It can also output or reset its current value.

All the channels must be declared; we give their names and the types of the values they can communicate. If a channel is used only for synchronisation, its declaration contains only its name. For example, *Register* outputs the current value through the channel *out*; it may also be reset through channel *reset*.

The declaration of a process is composed by its name and by its specification. A process may be explicitly defined or compound: defined in terms of other processes. An explicit process specification is formed by a sequence of process paragraphs and a distinguished nameless main action, which defines the process behaviour. We use Z to define the state; in our example, *RegSt* describes the state of the process *Register*: it contains the current *value* stored in the register.

Process paragraphs include Z paragraphs and declarations of (parametrised) actions. An action can be a schema, a guarded command, an invocation to another action, or a combination of these constructs using CSP operators.

The primitive action *Skip* does not communicate any value or changes the state: it terminates immediately. The action *Stop* deadlocks, and *Chaos* diverges; the only guarantee in both cases is that the state invariant is maintained.

The prefixing operator is standard. However, a guard construction is also available. For instance, if the condition p is *true*, the action $p \mathbin{\&} c?x \rightarrow A$ inputs a value through channel c and assigns it to x, and then behaves like A, which has the variable x in scope. If, however, p is *false*, the same action blocks.

$$
\begin{array}{lcl}
\mathsf{Program} & ::= & \mathsf{Par}^* \ \mathsf{CDecls}^* \ \mathsf{ProcDecl}^* \\
\mathsf{CDecls} & ::= & \mathbf{channel} \ \mathsf{CDecl} \\
\mathsf{CDecl} & ::= & \mathsf{SimpleCDecl} \mid \mathsf{SimpleCDecl}; \ \mathsf{CDecl} \\
\mathsf{SimpleCDecl} & ::= & \mathsf{N}^+ \mid \mathsf{N}^+ : \mathsf{Exp} \\
\mathsf{CSExp} & ::= & \{\!|\,|\!\} \mid \{\!|\,\mathsf{N}^+\,|\!\} \mid \mathsf{N} \mid \mathsf{CSExp} \cup \mathsf{CSExp} \mid \mathsf{CSExp} \cap \mathsf{CSExp} \\
& \mid & \mathsf{CSExp} \setminus \mathsf{CSExp} \\
\mathsf{ProcDecl} & ::= & \mathbf{process} \ \mathsf{N} \mathrel{\widehat=} \mathsf{ParProc} \\
\mathsf{ParProc} & ::= & \mathsf{Decl} \bullet \mathsf{Proc} \mid \mathsf{Proc} \\
\mathsf{Proc} & ::= & \mathbf{begin} \ \mathsf{PPar}^* \ \mathbf{state} \ \mathsf{SchemaExp} \ \mathsf{PPar}^* \bullet \mathsf{Action} \ \mathbf{end} \mid \mathsf{N} \\
& \mid & \mathsf{Proc}; \ \mathsf{Proc} \mid \mathsf{Proc} \Box \mathsf{Proc} \mid \mathsf{Proc} \sqcap \mathsf{Proc} \\
& \mid & \mathsf{Proc} \ [\![\ \mathsf{CSExp} \ ]\!] \ \mathsf{Proc} \mid \mathsf{Proc} \ |||\ \mathsf{Proc} \mid \mathsf{Proc} \setminus \mathsf{CSExp} \\
& \mid & \mathsf{Proc}[\mathsf{N}^+ := \mathsf{N}^+] \mid \mathsf{ParProc}(\mathsf{Exp}^+) \\
& \mid & \fatsemi_{\S} \ \mathsf{Decl} \bullet \mathsf{Proc} \mid \sqcap \mathsf{Decl} \bullet \mathsf{Proc} \\
& \mid & \|| \ \mathsf{Decl} \ [\![\ \mathsf{CSExp}]\!] \bullet \mathsf{Proc} \mid \ ||| \ \mathsf{Decl} \bullet \mathsf{Proc} \\
\mathsf{NSExp} & ::= & \{\,\} \mid \{\mathsf{N}^+\} \mid \mathsf{N} \mid \mathsf{NSExp} \cup \mathsf{NSExp} \mid \mathsf{NSExp} \cap \mathsf{NSExp} \\
& \mid & \mathsf{NSExp} \setminus \mathsf{NSExp} \\
\mathsf{PPar} & ::= & \mathsf{Par} \mid \mathsf{N} \mathrel{\widehat=} \mathsf{ParAction} \\
\mathsf{ParAction} & ::= & \mathsf{Decl} \bullet \mathsf{Action} \mid \mathsf{Action} \\
\mathsf{Action} & ::= & \mathsf{SchemaExp} \mid \mathsf{CSPAction} \mid \mathsf{Command} \mid \mathsf{N} \\
\mathsf{CSPAction} & ::= & \mathit{Skip} \mid \mathit{Stop} \mid \mathit{Chaos} \mid \mathsf{Comm} \rightarrow \mathsf{Action} \mid \mathsf{Pred} \ \& \ \mathsf{Action} \\
& \mid & \mathsf{Action}; \ \mathsf{Action} \mid \mathsf{Action} \Box \mathsf{Action} \mid \mathsf{Action} \sqcap \mathsf{Action} \\
& \mid & \mathsf{Action} \ [\![\ \mathsf{NSExp} \mid \mathsf{CSExp} \mid \mathsf{NSExp} \ ]\!] \ \mathsf{Action} \\
& \mid & \mathsf{Action} \ |[\![\mathsf{NSExp} \mid \mathsf{NSExp}]\!]| \ \mathsf{Action} \\
& \mid & \mathsf{Action} \setminus \mathsf{CSExp} \mid \mu \, \mathsf{N} \bullet \mathsf{Action} \mid \mathsf{ParAction}(\mathsf{Exp}^+) \\
& \mid & \fatsemi_{\S} \ \mathsf{Decl} \bullet \mathsf{Action} \mid \sqcap \mathsf{Decl} \bullet \mathsf{Action} \\
\mathsf{Comm} & ::= & \mathsf{N?N} \mid \mathsf{N!Expression} \mid \mathsf{N} \\
\mathsf{Command} & ::= & \mathsf{N} := \mathsf{Exp} \mid \mathbf{if} \ \mathsf{GActions} \ \mathbf{fi} \mid \mathbf{var} \ \mathsf{Decl} \bullet \mathsf{Action} \\
\mathsf{GActions} & ::= & \mathsf{Pred} \rightarrow \mathsf{Action} \mid \mathsf{Pred} \rightarrow \mathsf{Action} \Box \mathsf{GActions}
\end{array}
$$

Fig. 1. Executable *Circus* Syntax

The CSP operators of sequence, external and internal choice, parallelism, interleaving, hiding may also be used to compose actions. Communications and recursive definitions are also available. The process *Register* has a recursive behaviour: after its initialisation, it behaves like *RegCycle*, and then recurses. The action *RegCycle* is an external choice: values may be stored or accumulated, using channels *store* and *add*; the result may be requested using channel *result*, and output through *out*; finally, the register may be reset through channel *reset*.

The parallelism and interleaving operators are different from those of CSP. We must declare a synchronisation channel set, and, to avoid conflicts, two sets that partition the variables in scope: state components, and input and local variables. In a parallelism $A_1 \ [\![\ ns_1 \mid cs \mid ns_2 \ ]\!] \ A_2$, the actions A_1 and A_2 synchronise on the channels in the set cs. Both A_1 and A_2 have access to the initial values of

$$
\begin{array}{l}
\textbf{channel}\ store, add, out : \mathbb{N} \\
\textbf{channel}\ result, reset \\
\textbf{process}\ Register \mathrel{\widehat=}\ \textbf{begin state}\ RegSt \mathrel{\widehat=} [value : \mathbb{N}] \\
\quad RegCycle \mathrel{\widehat=} store?newValue \rightarrow value := newValue \\
\qquad\qquad \Box\ add?newValue \rightarrow value := value + newValue \\
\qquad\qquad \Box\ result \rightarrow out!value \rightarrow Skip \\
\qquad\qquad \Box\ reset \rightarrow value := 0 \\
\quad \bullet\ value := 0;\ (\mu\, X \bullet RegCycle;\ X)\ \textbf{end} \\
\\
\textbf{channel}\ read, write : \mathbb{N} \\
\textbf{process}\ SumClient \mathrel{\widehat=} \\
\quad \textbf{begin}\ ReadValue \mathrel{\widehat=} read?n \rightarrow reset \rightarrow Sum(n) \\
\qquad Sum \mathrel{\widehat=}\ n : \mathbb{N} \bullet (n = 0)\ \&\ result \rightarrow out?r \rightarrow write!r \rightarrow Skip \\
\qquad\qquad\qquad \Box\ (n \neq 0)\ \&\ add!n \rightarrow Sum(n-1) \\
\qquad \bullet\ \mu\, X \bullet ReadValue;\ X\ \textbf{end} \\
\textbf{chanset}\ RegAlphabet == \{\!|\ store, add, out, result, reset\ |\!\} \\
\textbf{process}\ Summation \mathrel{\widehat=} (SumClient\ [\![\ RegAlphabet\ ]\!]\ Register) \setminus RegAlphabet
\end{array}
$$

Fig. 2. A simple register

all variables in ns_1 and ns_2, but A_1 may modify only the values of the variables in ns_1, and A_2, the values of the variables in ns_2.

References to parametrised actions need to be instantiated. Actions may also be defined using assignment, guarded alternation, or variable blocks. Finally, in the interest of supporting a calculational approach to development, an action can be a Morgan's specification statement [9].

The CSP operators of sequence, external and internal choice, parallelism, interleaving, and hiding may also be used to compose processes. Furthermore, the renaming $P[oldc := newc]$ replaces all the references to channels *oldc* by the corresponding channels in *newc*, which are implicitly declared. Parametrised processes may also be instantiated.

In Figure 2, the process *SumClient* repeatedly receives a value n through channel *read*, interacts with *Register* to calculate the sum $\sum_{i=0}^{n} i$, and finally outputs this value through *write*. The process *Summation* is the parallel composition of *Register* and *SumClient*. They synchronise on the set of channels *RegAlphabet*, which is hidden from the environment: iterations with *Summation* can only be made through *read* and *write*.

Some other operators are available in *Circus*, but are omitted here for conciseness. The translation of these operators is either trivial or left as future work. They are discussed in Section 5.

3 JCSP

Since the facilities for concurrency in Java do not directly correspond with the idea of processes in CSP and *Circus*, we use JCSP, a library that provides a

model for processes and channels. This allows us to abstract from basic monitor constructs provided by Java. In JCSP, a process is a class that implements the `interface CSProcess{public void run();}`, where the method `run` encodes its behaviour. We present an `Example` process below.

```
import jcsp.lang.*; // further imports
class Example implements CSProcess {
    // state information, constructors, and auxiliary methods
    public void run { /* execution of the process */ } }
```

After importing the basic JCSP classes and any other relevant classes, we declare `Example`, which may have private attributes, constructors, and auxiliary methods. Finally, we must give the implementation of the method `run`.

Some JCSP interfaces represent channels: `ChannelInput` is the type of channels used to read objects; `ChannelOutput` is for channels used to write objects; and `AltingChannel` is for channels used in choices. Other interfaces are available, but these are the only ones used in our work.

The simplest implementation of a channel interface is that provided by the class `One2OneChannel`, which represents a point-to-point channel; multiple readers and writers are not allowed. On the other hand, `Any2OneChannel` channels allow many writers to communicate with one reader. For any type of channel, a communication happens between one writer and one reader only.

Mostly, JCSP channels communicate Java objects. For instance, in order to communicate an object `o` through a channel `c`, a writer process may declare `c` as a `ChannelOutput`, and invoke `c.write(o)`; a reader process that declares `c` as a `ChannelInput` invokes `c.read()`.

The class `Alternative` implements the choice operator. Although other types of choice are available, we use a fair choice. Only `AltingChannelInput` channels may be involved in choices. The code below reads from either channel `l` or `r`.

```
AltingChannelInput[] chs = new AltingChannelInput[]{l,r};
final Alternative alt = new Alternative(chs);
chs[alt.select()].read();
```

The channels `l` and `r` are declared in an array of channels `chs`, which is given to the constructor of the `Alternative`. The method `select` waits for one or more channels to become ready, makes an arbitrary choice between them, and returns an `int` that corresponds to the index of the chosen channel in `chs`. Finally, we read from the channel located at the chosen position of `chs`.

Parallel processes are implemented using the class `Parallel`. Its constructor takes an array of `CSProcesses` and returns a `CSProcess` that is the parallel composition of its process arguments. A `run` of a `Parallel` process terminates when all its component processes terminate. For instance, the code `(new Parallel(new CSProcess[]{P_1,P_2})).run();` runs two processes `P_1` and `P_2` in parallel. It creates the array of processes which will run in parallel, gives it to the constructor of `Parallel`, and finally, runs the parallelism.

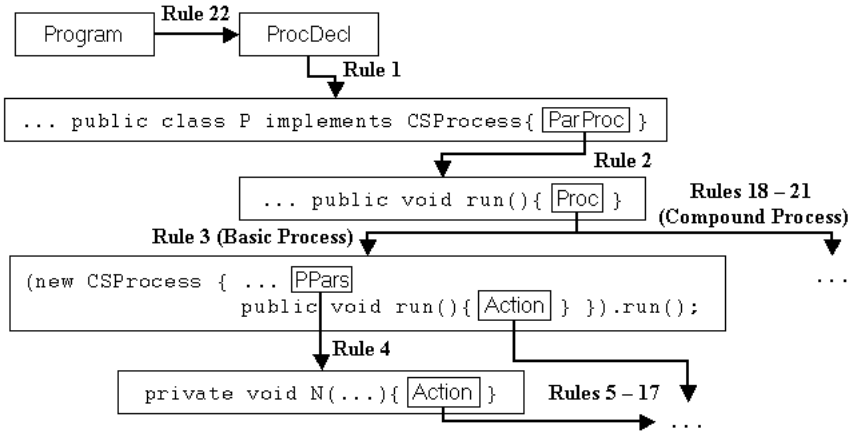

Fig. 3. Translation Strategy Overview

The CSP constructors *Skip* and *Stop* are implemented by the classes `Skip` and `Stop`. JCSP includes other facilities beyond those available in CSP; here we concentrate on those that are relevant for our work.

4 From Circus to JCSP

Our strategy for translating *Circus* programs considers each paragraph individually, and in sequence. In Figure 3, we present an overview of the translation strategy. First, for a given Program, we use a rule (Rule 22) that deals with the Z paragraphs and channel declarations. Each process declaration ProcDecl in the program is transformed into a new Java class (Rule 1). The next step (Rule 2) declares the class attributes, constructor, and its `run` method. Basic process definitions are translated (Rule 3) to the execution of a process whose private methods correspond to the translation (Rule 4) of actions of the original *Circus* process; the translation (Rules 5-17) of the main Action, which determines the body of the method `run`, and of the Action bodies conclude the translation of basic processes. Compound processes are translated using a separate set of rules (Rules 18-21) that combines the translations of the basic processes.

Only executable *Circus* programs can be translated: the technique in [**?**,18] can be used to refine specifications. Other restrictions are syntactic and can be enforced by a (mechanised) pre-processing; they are listed below.

- The *Circus* program is well-typed and well-formed.
- Paragraphs are grouped in the following order: Z paragraphs, channel declarations, and process declarations.
- Z paragraphs are axiomatic definitions of the form $v : T \mid v = e_1$, free types, or abbreviations.
- The only Z paragraphs inside a process declaration are axiomatic definitions of the above form.

- Variable declarations are of the form $x_1 : T_1;\ x_2 : T_2;\ \ldots;\ x_n : T_n$, and names are not reused.
- There are no nested external choices or nested guards.
- The synchronisation sets in the parallelisms are the intersection of the sets of channels used by the parallel actions or processes.
- No channel is used by two interleaved actions or processes.
- The types used are already implemented in Java.
- Only free types, abbreviations, and finite subsets of $\mathbb{N}$ and $\mathbb{Z}$ with equally spaced elements, are used for typing indexing variables of iterated operators.
- There are no multi-synchronisations or guarded outputs.

Axiomatic definitions can be used to define only constants. All types, abbreviations and free types, need a corresponding Java implementation. If necessary, the *Circus* data refinement technique should be used. In [10] we present rules to translate some forms of abbreviations and free types. Nested external choices and guarded actions can be eliminated with simple refinement laws.

The JCSP parallel construct does not allow the definition of a synchronisation channel set. For this reason, the intersection of the alphabets determines this set: if it is not empty, we have a parallelism; otherwise, we have actually an interleaving. JCSP does not have an interleaving construct; when possible we use the parallel construct instead.

Multi-synchronisation channels and guarded outputs are not implementable in JCSP. Before applying the translation strategy they must be removed applying refinement strategies as those presented in [18, 8].

The types of indexing variables in iterated operators are considered to be finite, because their translation uses loops. A different approach in the translation could make it possible to remove this restriction.

The output of the translation is Java code composed of several class declarations that can be split into different files and allocated in packages. For each program, we require a project name `proj`. The translation generates six packages: `proj` contains the main class, which is used to execute the system; `proj.axiomaticDefinitions` contains the class that encapsulates the translation of all axiomatic definitions; the processes are declared in the package `proj.processes`; `proj.typing` contains all the classes that implement types; and `proj.util` contains all the utility classes used by the generated code. For example, class `RandomGenerator` is used to generate random numbers; it is used in the implementation of internal choice.

The translation uses a channel environment δ. For each channel c, it maps c to its type, or to *sync*, if c is a synchronisation channel. We consider δ to be available throughout the translation.

For each process, two environments store information about channels: ν and ι for visible and hidden channels. They both map channel names to an element of $ChanUse ::= I \mid O \mid A$. The constant I is used for input channels, O for output channels, and A for input channels that take part in external choices. Synchronisation channels must also be associated to one of these constants, since every JCSP channel is either an input or an output channel. If a channel c

is regarded as an input channel in a process P, then it must be regarded as an output channel in any process parallel to P, and vice-versa.

The function *JType* defines the Java type corresponding to each of the used *Circus* types; and *JExp* translates expressions. The definitions of these functions are simple; for conciseness, we omit them. For example, we have that $JType(\mathbb{N}) =$`Integer`, and $JExp(x > y) =$ `x.intValue() > y.intValue()`.

This section is organised as follows: the rules of translation of processes declarations are presented in Section 4.1. Section 4.2 presents the translation of the body of basic processes, which is followed by the translation of the CSP actions (Section 4.3), and commands (Section 4.4). The translation of compound processes is presented in Section 4.5. Finally, Section 4.6 presents how to run the program. For conciseness, we omit some of the formal definitions of our translation strategy. They can be found in [10].

4.1 Processes Declarations

Each process declaration is translated to a Java class that implements the JCSP interface `jcsp.lang.CSProcess`. For a process P in a project named *proj*, we declare a Java class `P` that imports the Java utilities package, the basic JCSP package, and all the project packages.

Rule 1 $[\![\textbf{process}\ P \mathrel{\widehat=} ParProc]\!]^{ProcDecl}\ proj =$

```
package proj.processes; import java.util.*;
import jcsp.lang.*; import proj.axiomaticDefinitions.*;
import proj.typing.*; import proj.util.*;
public class P implements CSProcess { [[ ParProc ]]^ParProc P }
```

The function $[\![\_]\!]^{ProcDecl}$ takes a *Circus* process declaration and a project name to yield an Java class definition; our rule defines this function. The body of the class is determined by the translation of the paragraphs of P.

As an example, we translate *Register*, *SumClient*, and *Summation* (Figure 2); the resulting code is in [11]. The translation of *Register* is shown below; we omit package and import declarations.

```
public class Register implements CSProcess
    { [[ begin ... • value := 0; (μ X • ... ) end ]]^ParProc Register }
```

The translation the body of a parametrised process is captured by the function $[\![\_]\!]^{ParProc} : \mathsf{ParProc} \pfun \mathsf{N} \pfun \mathsf{JCode}$.

Rule 2 $[\![D \bullet P]\!]^{ParProc}\ N =$ (*ParDecl D*) (*VisCDecl ν*) (*HidCDecl ι*)

```
public N(ParArgs D, VisCArgs ν) {
    (MAss (ParDecl D) (ParArgs D))
    (MAss (VisCDecl ν) (VisCArgs ν))
    HidCC ι }
public void run(){ [[ P ]]^Proc }
```

The process parameters D are declared as attributes: for each $x : T$, the func-

tion *ParDecl* yields a declaration `private` (*JType T*) `x;`. The visible channels are also declared as attributes: for each channel c, with use t, *VisCDecl* gives `private` (*TypeChan t*) `c;`, where *TypeChan t* gives `ChannelInput` for $t = I$, `ChannelOutput` for $t = O$, and `AltingChannelInput` for $t = A$. For *Register*, we have declarations for the channels in the set *RegAlphabet*.

```
private AltingChannelInput store;...; ChannelOutput out; ...;
```

Hidden channels are also declared as attributes, but they are instantiated within the class. We declare them as `Any2OneChannel`, which can be instantiated. The process *Summation* hides all the channels in the set *RegAlphabet*. For this reason, within `Summation` they are declared to be of type `Any2OneChannel`.

The constructor receives the processes parameters and visible channels as arguments (*ParArgs D* and *VisCArgs* ν generates fresh names). The arguments are used to initialise the corresponding attributes (*MAss* (*ParDecl D*) (*ParArgs D*) and *MAss* (*VisCDecl* ν) (*VisCArgs* ν)), and hidden channels are instantiated locally (*HidCC* ι). In our example, we have the result below.

```
public Register (AltingChannelInput newstore, ...)
    { this.store = newstore; ... }
```

For *Summation*, we have the instantiation of all channels in the set *RegAlphabet*. For instance, `this.store = new Any2OneChannel();` instantiates *store*.

Finally, the method `run` implements the process body translated by $[\![\_]\!]^{Proc}$. In our example, we have `public void run(){`$[\![$**begin** ... **end**$]\!]^{Proc}$ `}`. For a non-parametrised process, like *Register*, we actually do not use Rule 1, but a simpler rule. The difference between the translation of parametrised and non-parametrised processes are the attributes corresponding to parameters.

4.2 Basic Processes

Each process body is translated by $[\![\_]\!]^{Proc} : \mathsf{Proc} \nrightarrow \mathsf{JCode}$ to an execution of an anonymous inner class that implements `CSProcess`. Inner classes are a Java feature that allows classes to be defined inside classes. The use of inner classes allows the compositional translation even in the presence of nameless processes.

Basic processes are translated as follows.

Rule 3 $[\![$**begin** *PPars*$_1$ **state** *PSt PPars*$_2$ $\bullet$ *A*$]\!]^{Proc} =$

```
(new CSProcess(){ (StateDecl PSt) ([[PPars1 PPars2]]^PPars)
                    public void run(){ [[ A ]]^Action }}).run();
```

The inner class declares the state components as attributes (*StateDecl PSt*). Each action gives rise to a private method ($[\![PPars_1\ PPars_2]\!]^{PPars}$). The body of `run` is the the translation of the main action *A*. Our strategy ignores any existing state invariants, since they have already been considered in the refinement of the process. It is kept in a *Circus* program just for documentation purposes.

As an example, we present the translation of the body of *Register*. For conciseness, we name its paragraphs *PPars*, and its main action *Main*.

$$\texttt{(new CSProcess()\{ private Integer value; } [\![PPars]\!]^{PPars} \texttt{ public void run() \{} [\![Main]\!]^{Action} \texttt{\} \}).run();}$$

The function $[\![\_]\!]^{PPars} : \mathsf{PPar}^* \pfun \mathsf{JCode}$ translates the paragraphs within a *Circus* process, which can either be axiomatic definitions, or (parametrised) actions. The translation of an axiomatic definition $v : T \mid v = e_1$ is a method `private` $(JType\ T)$ `v(){return` $(JExp\ e_1)$`;}`. Since the paragraphs of a process p can only be referenced within p, the method is declared `private`. We omit the relevant rule, and a few others in the sequel, for conciseness.

Both parametrised actions and non-parametrised actions are translated into private methods. However, the former requires that the parameters are declared as arguments of the new method. The reason for the method to be declared `private` is the same as that for the axiomatic definitions above.

Rule 4 $[\![N \mathrel{\widehat=} (D \bullet A)\ PPars]\!]^{PPars} =$

$$\texttt{private void N(}ParArgs\ D\texttt{)\{} [\![A]\!]^{Action} \texttt{\}} [\![PPars]\!]^{PPars}$$

The function *ParArgs* declares an argument for each of the process parameters. The body of the method is defined by the translation of the action body.

For instance, the translation of action *RegCycle* generates the following Java code. We use *body* to denote the body of the action.

$$[\![RegCycle \mathrel{\widehat=} body]\!]^{PPars} = \texttt{private void RegCycle()\{} [\![body]\!]^{Action} \texttt{\}}$$

The function $[\![\_]\!]^{Action} : \mathsf{Action} \pfun \mathsf{JCode}$ translates CSP actions and commands.

4.3 CSP Actions

In the translation of each action, the environment λ is used to record the local variables in scope in the translation of parallel and recursive actions. For each variable, λ maps its name to its type. Besides, as for processes, we have channel environments ν and ι to store information about how each channel is used.

The translations of *Skip* and *Stop* use basic JCSP classes: *Skip* is translated to `(new Skip()).run();`, and *Stop* is translated to `(new Stop()).run();`. *Chaos* is translated to an infinite loop `while(true){};`, which is a valid refinement of *Chaos*. For input communications, we declare a new variable whose value is read from the channel. A cast is needed, since the type of the objects transmitted through the channels is `Object`; we use the channel environment δ.

Rule 5 $[\![c?x \then Act]\!]^{Action} = \texttt{\{ } t\ \texttt{x = (}t\texttt{)c.read();} [\![Act]\!]^{Action} \texttt{\}}$

where $t = JType(\delta\ c)$. □

For instance, the communication *add?newValue* used in the action *RegCycle* is translated to `Integer newValue = (Integer)add.read();`

An output communication is easily translated as follows.

Rule 6 $[\![\, c!x \rightarrow Act \,]\!]^{Action} =$ `c.write(x);` $[\![\, Act \,]\!]^{Action}$

For synchronisation channels, we need to know whether it is regarded as an input or output channel; this information is retrieved either from ν or ι.

Rule 7 $[\![c \rightarrow Act]\!]^{Action} =$ `c.read();`
provided $\nu\ c \in \{ I, A \} \vee \iota\ c \in \{ I, A \}$ □

Rule 8 $[\![c \rightarrow Act]\!]^{Action} =$ `c.write(null);`
provided $\nu\ c = O \vee \iota\ c = O$ □

For example, in the process *SumClient*, the action $reset \rightarrow Sum(n)$ is translated to `reset.write(null);`, followed by the translation of $Sum(n)$. Within *Register*, the translation of *reset* is `reset.read();`. The difference is because *reset* is an output channel for *SumClient*, and an input channel for *Register*.

Sequential compositions are translated to a Java sequential composition.

Rule 9 $[\![\, A_1; \ \ldots\ ;\ A_n \,]\!]^{Action} = [\![\, A_1 \,]\!]^{Action}$; ... ; $[\![\, A_n \,]\!]^{Action}$

The translation of external choice uses the corresponding `Alternative` JCSP class; all the initial visible channels involved take part.

Rule 10 $[\![A_1 \,\Box\, \ldots \,\Box\, A_n]\!]^{Action} =$

```
Guard[] g = new Guard[]{ICAtt A1,...,ICAtt An};
final Alternative alt = new Alternative(g);
(DeclCs (ExIC A1) 0) ... (DeclCs (ExIC An) (#(ExIC An−1)))
switch(alt.select())
    { Cases (ExIC A1) A1 ... Cases (ExIC An) An }
```

provided $A_1, \ldots, A_n$ *are not guarded actions* g_i & A_i. □

In Figure 4 we present the translation of the body of *RegCycle*. It declares an array containing all initial visible channels of the choice (1). The function *ICAtt* returns a `,`-separated list of all initial visible channels of an action; informally, these are the first channels through which the action is prepared to communicate. The array is used in the instantiation of the `Alternative` process (2). Next, an `int` constant is declared for each channel (3). The function *DeclCs* returns a `;`-separated list of `int` constant declarations. The first constant is initialised with 0, and each subsequent constant with the previous constant incremented by one. Finally, a choice is made, and the chosen action executed. We use a `switch` block (4); the body of each `case` is the translation of the corresponding action (5); the function *Cases* takes the initial visible channel as argument (*ExIC*).

For guarded actions $\Box_i\, g_i$ & A_i, we have to declare an array `g` of boolean *JExp* g_i. We use this array in the selection `alt.select(g)`. Each unguarded action A_i can be easily refined to *true* & A_i.

If the guards are mutually exclusive, we can apply a different rule to obtain an `if-then-else`. This simplifies the generated code, and does not require the guarded actions to be explored in the translation of the external choice.

```
Guard[] guards = new Guard[]{store,add,result,reset};                    (1)
final Alternative alt = new Alternative(guards);                         (2)
final int C_STORE = 0; ... ; final int C_RESET = 3;                      (3)
switch(alt.select())                                                     (4)
    { case C_STORE:{...} break; ...; case C_RESET:{...} break; }         (5)
```

Fig. 4. Example of External Choice Translation

The translation of an internal choice chooses a random number between 1 and n. It uses the static method `generateNumber` of class `RandomGenerator`. Finally, it uses a `switch` block to choose and run the chosen action.

Rule 11 $[\![A_1 \sqcap \ldots \sqcap A_n]\!]^{Action} =$

`switch(RandomGenerator.generateNumber(1,n))`
`{case 1:{` $[\![A_1]\!]^{Action}$ `}break;` $\ldots$ `case n:{` $[\![A_n]\!]^{Action}$ `}break;}`

To translate a parallelism, we define an inner class for each parallel action, because the JCSP `Parallel` constructor takes an array of processes as argument. To deal with the partition of the variables, we use auxiliary variables to make copies of each state component. The body of each branch is translated and each reference to a state component is replaced with its copy. After the parallelism, we merge the values of the variables in each partition.

Local variables need to be copied as well, but since they are not translated to attributes, as state components are, they cannot be directly accessed in the inner classes created for each parallel action. For this reason, their copies are not initialised when declared; they are initialised in the constructor of each parallel action. Their initial values are given to the constructor as arguments.

The names of the inner classes are defined in the translation. To avoid clashes, we use a fresh index *ind* in the name of inner classes and local variables copies. In the following rule, *LName* and *RName* stand for the names of the classes that implement A_1 and A_2. We omit *RName*, which is similar to *LName*.

The function *IAuxVars* declares and initialises an auxiliary variable for each state component in the partition of A_1. Next, *DeclLcVars* declares one copy of each local variable; the initial values are taken by the constructor (*LcVarsArgs*). In the body of the constructor, the function *ILcVars* initialises each local variable with the corresponding value received as argument. The body of the method `run` is the translation of the action. The function *RenVars* is used to replace occurrences of the state components and variables in scope with their copies.

After the conclusion of the declaration of the inner class *LName*, we create an object of *LName*. A similar approach is taken in the translation of A_2 to *RName* and an object creation. The next step is to run the parallelism. Afterwards, a merge retrieves the final values of the state components and the variables in scope from their copies (*MergeVars*).

```
class ParLBranch_0 implements CSProcess {                    (1)
    public Integer aux_l_x_0 = x;                            (2)
    public Integer aux_l_y_0;                                (3)
    public ParLBranch_0(Integer y) { this.aux_l_y_0 = y; }   (4)
    public void run() { aux_l_x_0 = new Integer(0); } }      (5)
CSProcess l_0 = new ParLBranch_0(y);                         (6)
\* Right-hand side of the parallelism*\                     (7)
CSProcess[] procs_0 = new CSProcess[]{l_0,r_0};              (8)
(new Parallel(procs_0)).run ();                              (9)
x = ((ParLBranch_0)procs_0[0]).aux_l_x_0;                   (10)
y = ((ParRBranch_0)procs_0[1]).aux_r_y_0;                   (11)
```

Fig. 5. Example of Parallelism Translation

Rule 12 $[\![A_1 \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, A_2]\!]^{Action} =$

```
class LName implements CSProcess {
    (IAuxVars (ns1 \ (dom λ)) ind L) (DeclLcVars λ ind L)
    public LName((LcVarsArg λ)) { ILcVars λ ind L }
    public void run()
        { RenVars [[ A1 ]]^Action (ns1 ∪ (dom λ)) ind L }}
CSProcess l_ind = new LName(JList (ListFirst λ));
\\class RName declaration, process r_ind instantiation
CSProcess[] procs_ind = new CSProcess[]{ l_ind,r_ind };
(new Parallel(procs_ind)).run();
(MergeVars LName ns1 ind L) (MergeVars RName ns2 ind R)
```

where $LName =$ `ParLBranch_`ind *and* $RName =$ `ParRBranch_`ind

For instance, we present the translation of $x := 0 \,[\![\, \{x\} \mid \emptyset \mid \{y\} \,]\!]\, y := 1$ in Figure 5. We consider that the action occurs within a process with one state component $x : \mathbb{N}$, and that there is one local variable $y : \mathbb{N}$ in scope.

The state component x is declared in the left partition of the parallelism. For this reason, the class `ParLBranch_0` has two attributes: one corresponding to the state component x (2) and one corresponding to the local variable y (3), whose initial value is received in the constructor (4). The body of the method `run` (5) replaces all the occurrences of `x` by its copy `aux_l_x_0`. This concludes the declaration of the class `ParLBranch_0`, which is followed by the creation of an object `l_0` of this class (6). For conciseness, we omit the declaration of the class related to the right-hand side of the parallelism (7). Its declaration, however, is very similar to the left-hand side: its only auxiliary variable `aux_l_y_0` is declared and initialised as in class `ParLBranch_0`; the body of method `run` is the assignment `aux_r_y_0 = new Integer(1);`. Finally, after running the parallelism (8,9), the final value of `x` is that of its left branch copy (10), and the final value of `y` is that of its right branch copy (11).

```
value:=new Integer(0);                                              (1)
class I_0 implements CSProcess {                                    (2)
    public Integer aux_l_value_0;                                   (3)
    public I_0(Integer value){ this.aux_l_value_0 = value; }        (4)
    public void run() {                                             (5)
        RegCycle();                                                 (6)
        I_0 i_0_1 = new I_0(aux_l_value_0); i_0_1.run();            (7)
        aux_l_value_0 = i_0_1.aux_l_value_0; } };                   (8)
I_0 i_0_2 = new I_0(value); i_0_2.run();                            (9)
value = i_0_2.aux_l_value_0;                                       (10)
```

Fig. 6. Example of Recursion Translation

If we have a *Circus* action invocation, all we have to do is to translate it to a method call. If no parameter is given, the method invocation has no parameters. However, if any parameter is given, we use a Java expression corresponding to each parameter in the method invocation. In our example, $Sum(n)$ and $Sum(n-1)$ translate to `Sum(n);` and `Sum(new Integer(n.intValue()-1));`.

In order to avoid the need of indexing recursion variables, we also use inner classes to declare the body of recursions. As for parallelism, this requires the use of copies of local variables, which are declared as attributes of the inner class, and initialised in its constructor with the values given as arguments. The `run` method of this new inner class executes the body of the recursion, instantiates a new object of this class, where the recursion occurs, and executes it.

Rule 13 $[\![\mu\, X \bullet A(X)]\!]^{Action} =$

```
class I_ind implements CSProcess {
    DeclLcVars λ ind L
    public I_ind(LcVarsArg λ) { ILcVars λ ind L }
    public void run(){
        RenVars [| A(RunRec ind) |]^Action (dom λ) ind L}};
(RunRec ind)
```

The function *RunRec* instantiates a recursion process, invokes its `run` method, and finally collects the values of the auxiliary variables. For the same reason as for the translation of parallelism, we use a fresh index in the name of the inner class created for the recursion. Besides, since we are also using a inner class to express the recursion, the local variables must be given to the constructor of this inner class, and their final values retrieved after the execution of the recursion.

For instance, in Figure 6, we present the translation of the main action of process *Register*. First, we initialise `value` with 0 (1). Next, we declare the class `I_0`, which implements the recursion. It has a copy of the state component `value` as its attribute (3), which is initialised in the constructor (4). The method `run` calls the method `RegCycle` (6), instantiates a new recursion (7), and executes

it (8); this concludes the declaration of the recursion class. Next, we instantiate an object of this class, and execute it (9). Finally, we retrieve the final `value`.

The translation of parametrised action invocations also makes use of inner classes. Each of the local variables in scope has a corresponding copy as an attribute of the new class; the action parameters are also declared as attributes of the new class; both local variable copies attributes and parameters are initialised within the class constructor with the corresponding values given as arguments. The `run` method of the new class executes the parametrised action. However, the references to the local variables are replaced by references to their copies. Next, the translation creates an object of the class with the given arguments, and calls its `run` method. Finally, it restores the values of the local variables.

The translation of iterated sequential composition is presented below.

Rule 14 $[\![⨟ x_1 : T_1; \ldots; x_n : T_n \bullet Act]\!]^{Action} =$

InstActions `pV_`*ind* $(x_1 : T_1; \ldots; x_n : T_n)$ *Act ind*

```
for(int i = 0; i < pV_ind.size(); i++)
    { ((CSProcess)pV_ind.elementAt(i)).run(); }
```

The function *InstActions* declares an inner class `I_`*ind* that implements the action *Act* parametrised by the indexing variables. Then, it creates a vector `pV_`*ind* of actions using a nested loop over the possible values of each indexing variable: for each iteration, an object of `I_`*ind* is created using the current values of the indexing variables, and stored in `pV_`*ind*. Finally, each action within `pV_`*ind* is executed in sequence.

The translation of iterated internal choice uses the `RandomGenerator` to choose a value for each indexing variable. Then, it instantiates an action using the chosen values, and runs it.

4.4 Commands

Single assignments are directly translated to Java assignments.

Rule 15 $[\![x := e]\!]^{Action} =$ `x =` *(JExp e)*`;`

Variable declarations only introduce the declared variables in scope.

Rule 16 $[\![\textbf{var } x_1 : T_1; \ldots; x_n : T_n \bullet Act]\!]^{Action} =$

`{` *(JType* T_1*)* `x_1;` `...;` *(JType* T_n*)* `x_n;` $[\![Act]\!]^{Action}$ `}`

Alternations(**if_fi**) are translated to `if-then-else` blocks. The possible nondeterminism is removed by choosing the first *true* guard. If none of the guards is *true*, the action behaves like *Chaos* (`while(true){}`).

Rule 17 $[\![\textbf{if } g_1 \rightarrow A_1 \,\square\, \ldots \,\square\, g_n \rightarrow A_n \textbf{ fi}]\!]^{Action} =$

`if(`*JExp* g_1`){` $[\![A_1]\!]^{Action}$ `}...else if(`*JExp* g_n`){` $[\![A_n]\!]^{Action}$ `}`
`else { while(true){} }`

At this point, we are able to translate basic process. By way of illustration, Figure 7 presents a skeleton of the complete translation of process *Register*.

```
// Package declaration and imports (Rule 1)
public class Register implements CSProcess {
    private AltingChannelInput store; ...
    public Register (AltingChannelInput newstore, ...) { ... }
    public void run(){
        (new CSProcess(){
            private Integer value;
            private void RegCycle(){
                Guard[] guards = new Guard[]{store,add,result,reset};
                final Alternative alt = new Alternative(guards);
                final int C_STORE = 0; ...; final int C_RESET = 3;
                switch(alt.select()) { case C_STORE: { ... } break;
                                       case C_ADD: { ... } break;
                                       case C_RESULT: { ... } break;
                                       case C_RESET: { ... } break; } }
            public void run() { /* Figure 6 */ } }).run(); } }
```

Fig. 7. Translation of Process Register

4.5 Compound Processes

We now concentrate in the translation of the processes that are defined in terms of other processes. At this stage, we are actually translating the body of some process (Figure 3). This means, we are translating the body of its method `run`.

For a single process name N, we must instantiate the process `N`, and then, invoke its `run` method. The visible channels of the process are given as arguments to the process constructor. The function *ExtChans* returns a list of all channel names in the domain of the environment ν.

Rule 18 $[\![N]\!]^{Proc} =$ `(new CSProcess(){`
`  public void run(){(new N(`*ExtChans* ν`)).run();}`
`}).run();`

The invocation of (parametrised) processes is translated to a new inner class. It runs the parametrised process instantiated with the given arguments. The new classes names are also indexed by a fresh *ind* to avoid clashes.

The sequential composition of processes is also easily translated to the sequential execution of each process.

Rule 19 $[\![P_1 ; \ldots ; P_n]\!]^{Proc} = [\![P_1]\!]^{Proc}$ `;` $\ldots$ `;` $[\![P_n]\!]^{Proc}$

External choice has a similar solution to that presented for actions. The idea is to create an alternative in which all the initial channels of both processes, that are not hidden, take part. However, all auxiliary functions used in the previous definitions take actions into account. All we have to do is use similar functions that take processes into account.

As the internal choice for actions, the internal choice $P_1 \sqcap \ldots \sqcap P_n$ for processes randomly chooses a process, and then, starts to behave as such. Its definition is very similar to the corresponding one for actions.

The translation of parallelism executes a `Parallel` process. This process executes all the processes that are elements of the array given as argument to its constructor in parallel. In our case, this array has only two elements: each one corresponds to a process of the parallelism. Furthermore, the translation of parallelism of processes does not have to take into account variable partitions.

Rule 20 $[\![P_1 \,[\![\, cs \,]\!]\, P_2]\!]^{Proc} =$

```
(new CSProcess(){ public void run() {   new Parallel(
   new CSProcess[]{ [[ P1 ]]^Proc , [[ P2 ]]^Proc }).run();}}).run();
```

It is important to notice that, when using JCSP, the intersection of the alphabets determines the synchronisation channels set. For this reason, *cs* may be ignored.

The renaming operation $P[x_1, \ldots, x_n := y_1, \ldots, y_n]$ is translated by replacing all the `x_i`s by the corresponding `y_i`s in the translated Java code of P.

As for actions, the iterated operators are translated using `for` loops. The same restrictions apply for processes. The first iterated operator on processes is the sequential composition $\fatsemi$. As for actions, we use an auxiliary function to create a vector of processes, and execute in sequence each process within this vector. The iterated internal choice chooses a value for each indexing variable, and runs the process with the randomly chosen values for the indexing variables.

The translation of iterated parallelism of processes are simpler than that of actions, since we do not need to deal with partitions of variables in scope.

Rule 21 $[\![\, [\!]\, x_1 : T_1; \ldots; x_n : T_n \,[\![\, cs]\!] \bullet P]\!]^{Proc} =$

```
(new CSProcess(){
    public void run(){
      InstProcs pV_ind (x1 : T1; ...; xn : Tn) P ind
      CSProcess[] pA_ind = new CSProcess[pV_ind.size()];
      for (int i = 0; i < pV_ind.size(); i++)
          { pA_ind[i] = (CSProcess)pV_ind.get(i); }
      (new Parallel(pA_ind)).run(); } }).run();
```

It uses the function *InstProcs* to instantiate a vector `pV_`*ind* containing each of the processes obtained by considering each possible value of the indexing variables. Then, it transforms this `pV_`*ind* in an array `pA_`*ind*, which is given to the constructor of a `Parallel` process. Finally, we run the `Parallel` process.

The indexed operator translation uses array of channels. Its definition can be found in [10]. Furthermore, the translation of free types, abbreviations, generic channels, and further types of communications are also present in [10].

4.6 Running the program

The function $[\![\_]\!]^{Program}$ summarises our translation strategy. Besides the *Circus* program, this function also receives a project name, which is used to declare

the package for each new class. It declares the class that encapsulates all the axiomatic definitions (*DeclAxDefCls*), and translates all the declared processes.

Rule 22 $[\![Types\ \ AxDefs\ \ ChanDecls\ \ ProcDecls]\!]^{Program}\ proj =$
$(DeclAxDefCls\ \ proj\ \ AxDefs)\ ([\![ProcDecls]\!]^{ProcDecls}\ proj)$

In order to generate a class with a `main` method, which can be used to execute a given process, we use the function $[\![\_]\!]^{Run}$. This function is applied to a *Circus* process, and a project name. It creates a Java class named `Main`, which is created in the package *proj*. After the package declaration, the class imports the packages `java.util`, `jcsp.lang`, and all the packages within the project. The method `main` is defined as the translation of the given process.

For instance, in order to run the process *Summation*, we have to apply the function $[\![\_]\!]^{Run}$ to this process and give the project name `sum` as argument. This application results in the following Java code.

```
(new CSProcess() {
        public void run(){(new Summation()).run();} }).run();
```

For conciseness, we present only the body of the `main` method, and omit the package, import, class, and `main` method declarations.

5 Conclusion

The translation strategy presented in this work has been used to implement several programs, including a quite complex fire control system developed from its abstract centralised specification [10]. The application of the translation rules was straightforward; only human errors, which could be avoided if a translation tool were available, raised problems. The choice of JCSP was motivated by the local support of the JCSP implementors. Furthermore, the direct correspondence between many CSP and *Circus* constructs is a motivation for extending JCSP to support *Circus*, instead of creating another library from scratch.

Certainly, code generated by hand could be simpler. For instance, the translation of compound processes do not always need anonymous inner classes; they are used in the rules for generalisation purposes. However, our experiments have shown no significant improvement in performance after simplification.

In [3], Fischer formalises a translation from CSP-OZ to annotations of Java programs. A CSP-OZ specification is composed mostly by class definitions that model processes. They contain Z schemas that describe the internal state and its initialisation, and CSP processes that model the behaviour of the class. For each channel, an enable schema specifies when communication is possible, and an effect schema specifies the state changes caused by the communication.

In the translation, enable and effect schemas become pre and postconditions; the CSP part becomes trace assertions, which specify the allowed sequences of method calls; finally, state invariants become class invariants. The result is not an implementation of a CSP-OZ class, but annotations that support the verification

of a given implementation. The treatment of class composition is left as future work. Differently, our work supports the translation from *Circus* specifications, possibly describing the interaction between many processes, to correct Java code.

The translation from a subset of CSP-OZ to Java is also considered in [1], where a language called COZJava, which includes CSP-OZ and Java, is used. A CSP-OZ specification is first translated to a description of the structure of the final Java program, which still contains the original CSP processes and Z schemas; these are translated afterwards. The library that they use to implement processes is called CTJ [5], which is in many ways similar to JCSP. The architecture of the resulting Java program is determined by the architecture of CSP-OZ specifications, which keep communications and state update separate. As a consequence, the code is usually inefficient and complicated. It was this difficulty that motivated the design of *Circus*.

In *Circus*, communications are not attached to state changes, but are freely mixed as exemplified by the action *RegCycle* of process *Register*. As a consequence, the reuse of Z and CSP tools is not straightforward. On the other hand, *Circus* specifications can be refined to code that follow the usual style of programming in languages like occam, or JCSP, and are more efficient.

Due to JCSP limitations, we consider a restrict set of communications: non-typed inputs, outputs, and synchronisations. In [10], we treat generic channels and synchronisations $c.e$ over a channel c with expression e. Strategies to refine out the remaining forms of communication, multi-synchronisation, and guarded outputs are left as future work. A strategy to remove a special case of multi-synchronisation, in which it is not part of an external choice, is presented in [18].

JCSP itself restricts our strategy in the translation of parallelism. It does not support the definition of a synchronisation channel set: the intersection of the alphabets determines the synchronisation channels set.

We have considered the type of indexing variables of iterated operators to be finite. Furthermore, not all iterated operators are treated directly. The translation of iterated parallelism and interleaving of actions requires their expansion. For external choice, expansion is required for both the action and the process operator, due to the need to determine their initials. For conciseness, we omitted the *Circus* indexing operator, which expands the types of values communicated through the channels used in an action (or process) to include an index. Its simple translation involves arrays of channels; the rules can be found in [10].

The most important piece of future work is the implementation of a tool to support the translation strategy. In order to prove the soundness of such a tool, the proof of the translation rules presented here would be necessary. This, however, is a very complex task, as it involves the semantics of Java and *Circus*. We currently rely on the validation of the implementation of our industrial-scale case study [12] and on the fairly direct correspondence of JCSP and *Circus*.

Acknowledgements

We are grateful for the financial support of QinetiQ and the Royal Society. Jim Woodcock, Alistair McEwan, and Peter Welch provided valuable advice for our work.

References

1. A. L. C. Cavalcanti and A. C. A. Sampaio. From csp-oz to java with processes (extended version). Technical report, Centro de Informática/UFPE, 2000. Available at `http://www.cin.ufpe.br/~lmf`.
2. C. Fischer. Csp-oz: a combination of object-z and csp. In H. Bowmann and J. Derrick, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS'97)*, volume 2, pages 423 – 438. Chapman & Hall, 1997.
3. C. Fischer. *Combination and Implementation of Processes and Data: from CSP-OZ to Java*. PhD thesis, Fachbereich Informatik, Universität Oldenburg, Oldenburg - Germany, 2000.
4. A. J. Galloway. *Integrated Formal Methods with Richer Methodological Profiles for the Development of Multi-perspective Systems*. PhD thesis, University of Teeside, School of Computing and Mathematics, 1996.
5. G. Hilderink, J. Broenink, W. Vervoort, and A. Bakkers. Communicating java threads. In *Parallel Programming and Java Conference*, 1997.
6. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
7. C. A. R. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.
8. A. A. McEwan. *Concurrent Program Development*. PhD thesis, Oxford University Computing Laboratory, Oxford - UK, 2000. To appear.
9. Carroll Morgan. *Programming from Specifications*. Prentice-Hall, 1994.
10. M. V. M. Oliveira. A refinement calculus for *circus* - mini-thesis. Technical report.
11. M. V. M. Oliveira. From *circus* to jcsp - summation example source code, 2004. At http://www.cs.york.ac.uk/~marcel/circus/summation.pdf.
12. M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. Refining industrial scale systems in *circus*. In Ian East, Jeremy Martin, Peter Welch, David Duce, and Mark Green, editors, *Communicating Process Architectures*, volume 62 of *Concurrent Systems Engineering Series*, pages 281 – 309. IOS Press, 2004.
13. P.H.Welch, G.S.Stiles, G.H.Hilderink, and A.P.Bakkers. Csp for-java:multithreading for a ll.
14. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall Series in Computer Science. Prentice-Hall, 1998.
15. A. W. Roscoe, J. C. P. Woodcock, and L. Wulf. Non-interference through determinism. In D. Gollmann, editor, *ESORICS 94*, volume 1214 of *LNCS*, pages 33 – 54. Springer-Verlag, 1994.
16. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 2nd edition, 1992.
17. K. Taguchi and K. Araki. The state-based ccs semantics for concurrent z specification. In M. Hinchey and Shaoying Liu, editors, *International Conference on Formal Engineering Methods*, pages 283 – 292. IEEE, 1997.
18. J. C. P. Woodcock. Using *circus* for safety-critical applications. In *VI Brazilian Workshop on Formal Methods*, pages 1–15, Campina Grande, Brazil, 12th–14st October 2003.

19. J. C. P. Woodcock and A. L. C. Cavancanti. *Circus*: a concurrent refinement language. Technical report, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD UK, July 2001.
20. J. C. P. Woodcock and J. Davies. *Using Z – Specification, Refinement, and Proof*. Prentice-Hall, 1996.