Formal Development of Industrial-Scale Systems in *Circus*

Marcel Oliveira, Ana Cavalcanti, and Jim Woodcock

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

Abstract. *Circus* is a new notation that may be used to specify both data and behavioural aspects of a system, and has an associated refinement calculus. 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. Although a few case studies are already available in the literature, the industrial fire control system, whose refinement and implementation is discussed in this paper, is, as far as we know, the largest case study on the *Circus* refinement strategy.

Keywords: concurrency, refinement, program development, object-orientation.

1 Introduction

Languages like Z [24], VDM [11], Abstract State Machines[2], and B [1], use a model-based approach to specification, based on mathematical objects from set theory. 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 [9, 18] and CCS [12] 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 [22], Z with CSP [19], and Object-Z with CSP [6] are some examples. Our work is based on *Circus* [20, 4], which characterises systems as processes that combine constructs that describe data and control behaviour. The Z notation is used to define most of the data aspects, and CSP and Dijkstra's guarded-command language are used to define behaviour. The semantics of *Circus* is based on the *Unifying Theories of Programming* [10], a framework that unifies the science of programming across many different computational paradigms.

Circus, unlike the other combinations of data and behavioural aspects, supports refinement in a calculational style similar to [13]. A refinement strategy for *Circus* is presented in [4], with the complete development of a reactive buffer into a distributed implementation as an example, and extended in [16], where we discuss the refinement of a industrial scale fire control system in detail.

The main objective of this paper is to provide a translation strategy for implementing *Circus* programs in Java. The translation strategy is based on translation rules, which transform a *Circus* program into a Java program that uses the JCSP [17] library. These rules capture and generalise the approach that we took in the implementation of the fire control system. We believe that, with the results of this work, we provide empirical evidence of the power of expression of *Circus* and, principally, that the refinement strategy presented in [4] and the translation strategy presented in this paper are applicable to large industrial systems.

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 and can be used as a basis in the implementation of an automated translation to Java, which makes formal development based on *Circus* relevant in practice. We assume that, before applying the translation strategy, the specification of the system we want to implement has already been refined into a specification written in the executable subset of *Circus*, using the *Circus* refinement strategy presented in [4].

We describe *Circus* in the next section. JCSP is presented with some examples in Section 3 and the translation strategy is presented in Section 4. In Sections 5 and 6, we describe the fire control system, and discuss its refinement and implementation in JCSP. Finally, we present our conclusions and discuss future work in Section 7.

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 [21]. The boxed constructs in Figure 1, indicate the *Circus* constructs that are not part of the executable subset of *Circus*; the underlined constructs have a constrained syntax in this subset.

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 generic channel declaration **channel** $[T]\ c : T$ declares a family of channels c. In this declaration, $[T]$ is a parameter used to determine the type of the values that are communicated through channel c. We may introduce sets of channels in a **chanset** paragraph. Channels can also be declared using schemas that group channel declarations, but do not have a predicate part. This follows from the fact that the only restriction that may be imposed on a channel is the type of values that it communicates.

The declaration of a process is composed of its name and its specification. A process may be explicitly defined or compound. 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. Generic processes may also be declared and instantiated.

Indexed processes are particular to *Circus* specifications. The process $i : T \odot P$ behaves likes P but communicates via different channels. For each channel c in P, we have a new channel c_i that communicates pairs of values: the first element is an index of type T, and the second element is the original value that was communicated through c. Such processes may be instantiated: $(i : T \odot P)\lfloor e \rfloor$, communicates pairs of values where the first element is the value of the expression e.

Process paragraphs include Z paragraphs, declarations of (parameterised) actions, and sets of names. An action can be a schema, a guarded command, or an invocation of another action; actions can combined using CSP operators.

The primitive action *Skip* does not communicate any value, nor does it change 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.

$$
\begin{array}{lll}
\mathsf{Program} & ::= & \underline{\mathsf{CircusPar}^*} \\
\mathsf{CircusPar} & ::= & \mathsf{Par} \mid \textbf{channel}\ \mathsf{CDecl} \mid \boxed{\textbf{chanset}\ \mathsf{N} == \mathsf{CSExp}} \mid \mathsf{ProcDecl} \\
\mathsf{CDecl} & ::= & \mathsf{SimpleCDecl} \mid \mathsf{SimpleCDecl};\ \mathsf{CDecl} \\
\mathsf{SimpleCDecl} & ::= & \mathsf{N}^+ \mid \mathsf{N}^+ : \mathsf{Exp} \mid [\mathsf{N}^+]\mathsf{N}^+ : \mathsf{Exp} \mid \boxed{\mathsf{SchemaExp}} \\
\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} & ::= & \textbf{process}\ \mathsf{N} \mathrel{\widehat=} \mathsf{ProcDef} \mid \textbf{process}\ \mathsf{N}[\mathsf{N}^+] \mathrel{\widehat=} \mathsf{ProcDef} \\
\mathsf{ProcDef} & ::= & \underline{\mathsf{Decl} \bullet \mathsf{ProcDef}} \mid \underline{\mathsf{Decl} \odot \mathsf{ProcDef}} \mid \mathsf{Proc} \\
\mathsf{Proc} & ::= & \textbf{begin}\ \mathsf{PPar}^*\ \textbf{state}\ \mathsf{SchemaExp}\ \mathsf{PPar}^* \bullet \mathsf{Action}\ \textbf{end} \mid \mathsf{N} \\
 & \mid & \mathsf{Proc};\ \mathsf{Proc} \mid \mathsf{Proc} \mathbin{\Box} \mathsf{Proc} \mid \mathsf{Proc} \sqcap \mathsf{Proc} \\
 & \mid & \mathsf{Proc} \mathbin{|[} \mathsf{CSExp} \mathbin{]|} \mathsf{Proc} \mid \mathsf{Proc} \mathbin{|||} \mathsf{Proc} \mid \mathsf{Proc} \setminus \mathsf{CSExp} \\
 & \mid & \underline{(\mathsf{Decl} \bullet \mathsf{ProcDef})(\mathsf{Exp}^+)} \mid \mathsf{N}(\mathsf{Exp}^+) \\
 & \mid & \underline{(\mathsf{Decl} \odot \mathsf{ProcDef})\lfloor \mathsf{Exp}^+ \rfloor} \mid \mathsf{N}\lfloor \mathsf{Exp}^+ \rfloor \mid \mathsf{Proc}[\mathsf{N}^+ := \mathsf{N}^+] \mid \mathsf{N}[\mathsf{Exp}^+] \\
 & \mid & \fatsemi\ \mathsf{Decl} \bullet \mathsf{Proc} \mid \boxed{\Box\ \mathsf{Decl} \bullet \mathsf{Proc}} \mid \sqcap\ \mathsf{Decl} \bullet \mathsf{Proc} \\
 & \mid & |[\mathsf{CSExp}]|\ \mathsf{Decl} \bullet \mathsf{Proc} \mid\ |||\ \mathsf{Decl} \bullet \mathsf{Proc} \\
 & \mid & \fatsemi\ \mathsf{Decl} \odot \mathsf{Proc} \mid \boxed{\Box\ \mathsf{Decl} \odot \mathsf{Proc}} \mid \sqcap\ \mathsf{Decl} \odot \mathsf{Proc} \\
 & \mid & |[\mathsf{CSExp}]|\ \mathsf{Decl} \odot \mathsf{Proc} \mid\ |||\ \mathsf{Decl} \odot \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} \mid \boxed{\textbf{nameset}\ \mathsf{N} == \mathsf{NSExp}} \\
\mathsf{ParAction} & ::= & \mathsf{Decl} \bullet \mathsf{ParAction} \mid \mathsf{Action} \mid \mu\, \mathsf{N} \bullet \mathsf{ParAction} \\
\mathsf{Action} & ::= & \mathsf{SchemaExp} \mid \mathsf{CSPAction} \mid \mathsf{Command} \mid \mathsf{N} \mid \mathsf{Action}\,[\mathsf{N}^+ := \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} \mathbin{\Box} \mathsf{Action} \mid \mathsf{Action} \sqcap \mathsf{Action} \\
 & \mid & \mathsf{Action} \mathbin{[\![} \mathsf{NSExp} \mid \mathsf{CSExp} \mid \mathsf{NSExp} \mathbin{]\!]} \mathsf{Action} \\
 & \mid & \mathsf{Action} \mathbin{|\![} \mathsf{NSExp} \mid \mathsf{NSExp} \mathbin{]\!|} \mathsf{Action} \\
 & \mid & \mathsf{Action} \setminus \mathsf{CSExp} \mid \mathsf{ParAction}(\mathsf{Exp}^+) \\
 & \mid & \mu\, \mathsf{N} \bullet \mathsf{Action} \mid (\mu\, \mathsf{N} \bullet \mathsf{ParAction})(\mathsf{Exp}^+) \\
 & \mid & \fatsemi\ \mathsf{Decl} \bullet \mathsf{Action} \mid \boxed{\Box\ \mathsf{Decl} \bullet \mathsf{Action}} \mid \sqcap\ \mathsf{Decl} \bullet \mathsf{Action} \\
 & \mid & \boxed{|[\mathsf{CSExp}]|\ \mathsf{Decl} \bullet [\![\mathsf{NSExp}]\!]\ \mathsf{Action}} \\
 & \mid & \boxed{|||\ \mathsf{Decl} \bullet |\![\mathsf{NSExp}]\!|\ \mathsf{Action}} \\
\mathsf{Comm} & ::= & \underline{\mathsf{N}\ \ \mathsf{CParameter}^*} \mid \underline{\mathsf{N}\ \ [\mathsf{Exp}^+]\ \mathsf{CParameter}^*} \\
\mathsf{CParameter} & ::= & ?\mathsf{N} \mid \boxed{?\mathsf{N} : \mathsf{Pred}} \mid\ !\mathsf{Exp} \mid .\mathsf{Exp} \\
\mathsf{Command} & ::= & \boxed{\mathsf{N}^+ : [\mathsf{Pred}, \mathsf{Pred}]} \mid \{\mathsf{Pred}\} \mid \boxed{[\mathsf{Pred}]} \\
 & \mid & \mathsf{N}^+ := \mathsf{Exp}^+ \mid \textbf{if}\ \mathsf{GActions}\ \textbf{fi} \mid \textbf{var}\ \mathsf{Decl} \bullet \mathsf{Action} \\
\mathsf{GActions} & ::= & \mathsf{Pred} \rightarrow \mathsf{Action} \mid \mathsf{Pred} \rightarrow \mathsf{Action} \mathbin{\Box} \mathsf{GActions}
\end{array}
$$

Fig. 1. *Circus* Syntax

The guarded-prefixing operator is standard. 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. The instantiation of generic channels is also possible.

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

Fig. 2. A simple register

The CSP operators of sequence, external and internal choice, parallelism, interleaving, and hiding may also be used to compose actions; each of these has an iterated counterpart. 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*.

Parallelism follows the alphabetised approach adopted by [18]; we must declare a synchronisation channel set. In order to avoid conflicts, we must declare two sets that partition the variables in scope: state components, and input and local variables. In $A_1 \lbrack\!\lbrack ns_1 \mid cs \mid ns_2 \rbrack\!\rbrack A_2$, both A_1 and A_2 have access to the initial values of 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. Besides, the actions A_1 and A_2 synchronise on the channels in the set cs.

References to parameterised actions and recursions need to be instantiated. Actions may also be defined using recursion, assignment, guarded alternation, or variable blocks. Finally, a calculational approach to development is supported with the use of specification statements, assumptions, and coercions [13].

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

In Figure 2, the process *SumClient* repeatedly receives a value n through channel *read*, interacts with *Register* to calculate $\sum_{i=0}^{n} i$, and 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*.

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 Java `interface CSProcess{public void run();}`, where the method `run` encodes its behaviour. We present an `Example` process below.

```
import jcsp.lang.*;
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`, whose constructor takes an array of `CSProcesses` and returns the parallel composition of its process arguments. A `run` of a `Parallel` process terminates when all its component do. For instance, the code presented below executes two processes `P_1` and `P_2` in parallel.

```
(new Parallel(new CSProcess[]{P_1,P_2})).run();
```

It creates the array of processes which will run in parallel, gives it to the constructor of `Parallel`, and finally, runs the parallelism.

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 16) 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 5-12) 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 13-15 and 18-20) that combines the translations of the basic processes.

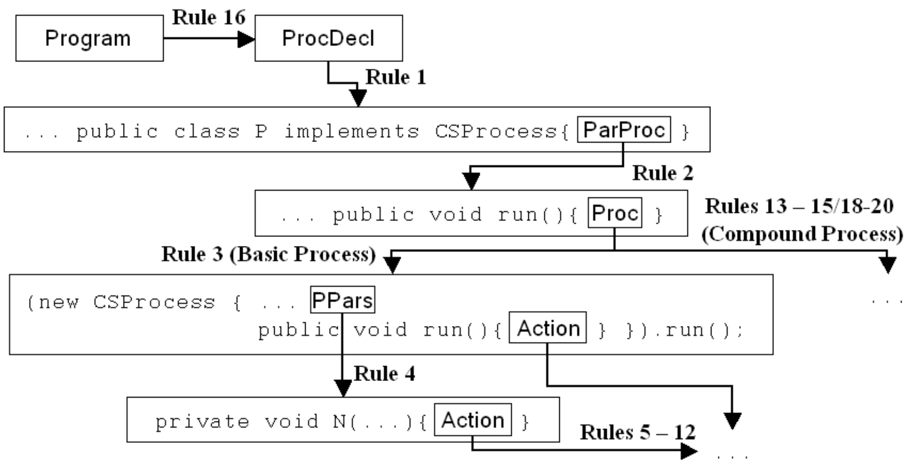
Program
Rule 16
ProcDecl
Rule 1
... public class P implements CSProcess{ ParProc }
Rule 2
... public void run(){ Proc }
Rules 13 – 15/18-20 (Compound Process)
Rule 3 (Basic Process)
(new CSProcess { ... PPars public void run(){ Action } }).run();
...
Rule 4
private void N(...){ Action }
Rules 5 – 12
...

Fig. 3. Translation Strategy Overview

Requirements. Only executable *Circus* programs can be translated: the technique in [4, 23] 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.
- Z paragraphs precede channel declarations, whcih precede process declarations.
- Z paragraphs are axioms $v : T \mid v = e_1$, free types, or abbreviations.
- The only Z paragraphs inside a process declaration are axioms as above.
- Variable declarations are of the form $x_1 : T_1;\ \cdots;\ x_n : T_n$; no name is reused.
- There are no nested external choices or nested guards.
- There are no nested declarations of parametrised and indexed processes.
- A synchronisation set is the intersection of the relevant sets of channels.
- No channel is used by two interleaved actions or processes.
- The types used are already implemented in Java.
- Only free types, abbreviations, and segments of $\mathbb{Z}$ are used for indexing variables.
- There are no guarded outputs.
- Multi-synchronisation channels are neither generic nor simple synchronisations.

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 [14] we present rules to translate some forms of abbreviations and free types. Nested external choices, guarded actions, and parametrised actions can be eliminated with simple refinement laws.

The JCSP parallel construct does not allow the definition of a synchronisation channel set, and so the intersection of the alphabets determines this set. Guarded outputs are not implementable in JCSP. Before applying the translation strategy they must be removed applying refinement strategies as in [23].

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. In order to simplify the definitions throughout this paper, we use a non-standard representation of a channel type. For instance, the generic channel declared as **channel** $[T]c : \mathbb{N} \times \mathbb{N}$ is represented in this environment as the mapping $c \mapsto ([T], [T, \mathbb{N}])$. The first list contains the typying variables and the second contains the types used in the declaration of the channel. Untyped channels are mapped to $([\,], [\mathit{Sync}])$.

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 $\mathit{ChanUse} ::= I \mid O \mid A$. 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*.

A type environment is also considered available in the translation: the environment $\tau : \mathrm{seq}\,\mathsf{Expression}$ lists all the types that are used in the *Circus* program which is being translated. This list includes all the basic types, free types, abbreviations, and possible types created for encapsulating multiple inputs and outputs.

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

Table 1 presents a summary of the environments that are used throughout the translation strategy. The environments λ, ς, and ω, have not yet been described; they are used in the translation of parallelism, generic channels, and multi-synchronisation, respectively; further details will be given as they are used.

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. In Section 4.7, we discuss the translation of synchronisation channels, which is followed by a discussion on the translation of the indexing operator. The translation

Table 1. Environments used in the Translation Strategy

Env	Description	Granularity
δ	Type of every channel in the system	
ν	Usage of visible channels (input, output, or alting)	Processes
ι	Usage of visible channels (input, output, or alting)	Processes
τ	Types used in the system	
λ	Type of every local variable and state component in scope	Actions
ς	Usage of channels (communication of values or not)	Processes
ω	Information about channel involved in multi-synchronisations	Processes

of generic channels and channels that take part in multi-synchronisation are presented in Sections 4.8 and 4.9, respectively. For conciseness, we omit some of the formal definitions of our translation strategy. They can be found in [14].

4.1 Processes Declarations

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

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

```
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 $[\![\_]\!]^{\mathit{ProcDecl}}$ 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 [14]. Besides the `package` and `import` declarations, the translation of *Register* yields `public class Register implements CSProcess {···}`; the body of this class is $[\![\,\mathbf{begin}\ \cdots\ \bullet\ \mathit{value} := 0;\ (\mu\, X \bullet \cdots)\ \mathbf{end}\,]\!]^{\mathit{ParProc}}$`Register`.

The translation the body of a parameterised process is captured by the function $[\![\_]\!]^{\mathit{ParProc}} : \mathsf{ParProc} \nrightarrow \mathsf{N} \nrightarrow \mathsf{JCode}$. The process parameters D are declared as attributes: for each $x : T$, the function *ParDecl* yields `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 $t = A$.

Rule 2 $[\![D \bullet P\,]\!]^{\mathit{ParProc}}\ N =$

```
(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 }
```

For *Register*, we have declarations for the channels in the set *RegAlphabet* correspond to `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 ($\mathit{ParArgs}\ D$ and $\mathit{VisCArgs}\ \nu$ generates fresh names). The arguments are used to initialise the corresponding attributes ($\mathit{MAss}\ (\mathit{ParDecl}\ D)\ (\mathit{ParArgs}\ D)$ and $\mathit{MAss}\ (\mathit{VisCDecl}\ \nu)\ (\mathit{VisCArgs}\ \nu)$), and hidden channels are instantiated locally ($\mathit{HidCC}\ \iota$). 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(){`$[\![\mathbf{begin}\ \cdots\ \mathbf{end}]\!]^{Proc}$ `}`. For a non-parameterised process, like *Register*, we actually do not use Rule 1, but a simpler rule. The difference between the translation of parameterised and non-parameterised processes are the attributes corresponding to parameters.

4.2 Basic Processes

Each process body is translated by $[\![\_]\!]^{Proc} : \mathsf{Proc} \rightarrow\!\!\!\!\!\!+\ \mathsf{JCode}$ 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 $[\![\mathbf{begin}\ \mathit{PPars}_1\ \mathbf{state}\ \mathit{PSt}\ \mathit{PPars}_2 \bullet A]\!]^{Proc} =$

`(new CSProcess(){` $(\mathit{StateDecl}\ \mathit{PSt})$ $([\![\mathit{PPars}_1\ \mathit{PPars}_2]\!]^{PPars})$
`public void run(){` $[\![A]\!]^{Act}$ `}}).run();`

The inner class declares the state components as attributes ($\mathit{StateDecl}\ \mathit{PSt}$). Each action gives rise to a private method ($[\![\mathit{PPars}_1\ \mathit{PPars}_2]\!]^{PPars}$). 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*.

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

The function $[\![\_]\!]^{PPars} : \mathsf{PPar}^{*} \rightarrow\!\!\!\!\!\!+\ \mathsf{JCode}$ translates the paragraphs within a *Circus* process, which can either be axiomatic definitions, or (parameterised) actions. The translation of an axiomatic definition $v : T \mid v = e_1$ is a method `private` $(\mathit{JType}\ T)$ `v(){return` $(\mathit{JExp}\ e_1)$`;}`. 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 parameterised actions and non-parameterised actions are translated into private methods; however, the former requires that the parameters are declared as arguments of the new method.

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

`private void N(`$\mathit{ParArgs}\ D$`){` $[\![A]\!]^{Act}$ `}` $[\![\mathit{PPars}]\!]^{PPars}$

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.

By way of illustration, the translation of *RegCycle* generates the Java code `private void RegCycle(){`$[\![\mathit{body}]\!]^{Act}$`}`. We use *body* to denote the body of the action. The function $[\![\_]\!]^{Act} : \mathsf{Action} \rightarrow\!\!\!\!\!\!+\ \mathsf{JCode}$ translates CSP actions and commands.

4.3 CSP Actions

For each action translation, the environment λ is records the local variables in scope and state components; it is used in the translation of parallel and recursive actions. For each variable and state component, λ maps its name to its type. We also have channel environments ν and ι to store information about 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 $[\![c?x \rightarrow Act]\!]^{Act} =$ `{` t `x = (`t`)c.read();` $[\![Act]\!]^{Act}$ `}`
where $t = JType(last\ (snd\ (\delta\ c)))$. □

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

An output communication $c!x$ is easily translated to `c.write(x);`. 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. If it is an input channel then it is translated to `c.read();`; otherwise it is translated to `c.write(null);`. For example, in *SumClient*, the action $reset \rightarrow \cdots$ is translated to `reset.write(null);`$\cdots$. On the other hand, in *Register*, the translation of *reset* is `reset.read();`.

Sequential compositions are simply translated to a Java sequential composition. The translation of external choice uses the corresponding `Alternative` JCSP class; all the initial visible channels involved take part.

Rule 6 $[\![A_1 \Box \cdots \Box A_n]\!]^{Act} =$
  `Guard[] g = new Guard[]{`$ICAtt\ A_1$`,`$\cdots$`,`$ICAtt\ A_n$`};`
  `final Alternative alt = new Alternative(g);`
  $(DeclCs\ (ExIC\ A_1)\ 0)\ \cdots\ (DeclCs\ (ExIC\ A_n)\ (\#(ExIC\ A_{n-1})))$
  `switch(alt.select()){`$Cases\ (ExIC\ A_1)\ A_1 \cdots\ Cases\ (ExIC\ A_n)\ A_n$`}`
provided $A_1, \cdots, A_n$ *are not guarded actions* $g_i\ \&\ A_i$. □

By way of illustration, we present below the translation of the body of *RegCycle*.

```
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)
```

It declares an array containing all initial channels of the choice (1). The function *ICAtt* returns a `,`-separated list of all initial channels of an action; 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 comma-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).

For guarded actions $\Box_i\ g_i\ \&\ A_i$, we declare an array `g` of boolean $JExp\ g_i$, which we use in the selection `alt.select(g)`. Unguarded actions A_i are refined to $true\ \&\ A_i$. If the guards are mutually exclusive, we can apply a different rule to obtain an `if-then-else`; this does not require the guarded actions to be explored in the translation of the external choice. The translation of $A_1 \sqcap \cdots \sqcap A_n$ chooses a random number between 1 and n (`RandomGenerator.generateNumber(1,n)`). This choice is used in a `switch` block to execute the translation of the the chosen action.

To translate a parallel construct, 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 parallel composition, 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, they cannot be directly accessed in the inner classes created for each parallel action. So their copies are not initialised when declared; they are initialised in the constructor of each parallel action, and the initial values are passed to the constructor. 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.

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 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 declaration of the inner class *LName*, we create an object of *LName* (the translation of A_2 is similar). Next, we run the parallel processes, and then merge the results to get the final values of the state and the variables in scope (*MergeVars*).

Rule 7 $[\![A_1 \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, A_2]\!]^{Act} =$

```
    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 ⟧^Act (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 $x := 0 \,[\![\, \{x\} \mid \emptyset \mid \{y\} \,]\!]\, y := 1$ below. The action is in a process with one state item $x : \mathbb{N}$, and local $y : \mathbb{N}$ in scope.

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

The state component x is declared in the left partition of the parallelism. For this reason, the class `ParLBranch_0` has two attributes: one copy of x and one copy of

y (2), whose initial value is received in the constructor (3). All the occurrences of `x` are replaced by its copy in the body of `run` (4). This concludes the declaration of `ParLBranch_0`, which is followed by the creation of an object of this class (5). For conciseness, we omit the declaration of the class related to the right-hand side of the parallelism (6). Its declaration is very similar to the left-hand side: the only copy `aux_l_y_0` is declared and initialised as in `ParLBranch_0`; the body `run` is the assignment `aux_r_y_0 = new Integer(1);`. After running the parallelism (7,8), the final values of `x` and `y` are those of their left (9) and right (10) copies, respectively.

A *Circus* action invocation is translated into a method call. If no parameter is given, the method invocation has no parameters. If any parameter is given, we use the java expression corresponding to it in the method invocation. For instance, $Sum(n-1)$ translate to `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 8 $[\![\mu X \bullet A(X)]\!]^{Act} =$

```
    class I_$ind$ implements CSProcess {
        $DeclLcVars\ \lambda\ ind\ L$
        public I_$ind$($LcVarsArg\ \lambda$) { $ILcVars\ \lambda\ ind\ L$ }
        public void run()
            {$RenVars\ [\![ A(RunRec\ ind\ \langle\rangle) ]\!]^{Act}\ (\mathrm{dom}\ \lambda)\ ind\ L$}};
    ($RunRec\ ind\ \langle\rangle$)
```

The function $RunRec$ instantiates a recursion process possibly using the given recursion arguments, invokes its `run` method, and finally collects the values of the auxiliary variables. We also use a fresh index in the name of the inner class created for the recursion to avoid name clashes. 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.

We present below the translation of the main action of process $Register$.

```
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)
```

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`.

We consider two cases for parametrised recursion: actions may be declared as parametrised recursion (*i.e.,* $A \mathrel{\widehat=} \mu X \bullet (y : \mathbb{N} \bullet A(X(y))))$, or actions may have instantiations of parametrised recursion (*i.e.,* $(\mu X \bullet (y, z : \mathbb{N} \bullet A(X(y, z))))(0, 0))$. In the first case, we simply consider it as a parametrised action, in which the recursion corresponds to invocation of the action name itself, using the given argument ($A(y)$). In the second case, we consider the translation of the recursion using an extension of the local variables environment in which the names of the recursion parameters y and z are included. The instantiation of the recursion in the recursion body uses the expressions list x, y used in the recursion call. However, the first execution of the recursion uses the expressions list $0, 0$ used in the recursion instantiation.

The translation of parameterised 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 parameterised 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 9 $[\![\,\fatsemi\, x_1 : T_1;\ \cdots;\ x_n : T_n \bullet Act]\!]^{Act} =$

```
    InstActions pV_ind (x_1 : T_1; ···; 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* parameterised by the indexing variables. 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*. Each action in `pV_`*ind* is executed in sequence. Iterated internal choice uses the `RandomGenerator` to choose a value for each indexing variable.

4.4 Commands

Single assignments are translated to `x = (`*JExp e*`);`. Multiple assignments where no expression mentions any variable on the left-hand side are implemented as a sequence of single assignments. Otherwise, we create an initial copy of every relevant variable, and use these copies in the assignment. The types of the copies are the same as the original variables; they are retrieved from the variables environment λ.

Rule 10 $[\![x_1, \cdots, x_n := e_1, \cdots, e_n]\!]^{Act} =$

```
    (JType (λ x_1)) aux_ind_x_1 = (JExp e_1); ··· ;
    (JType (λ x_n)) aux_ind_x_n = (JExp e_n);
    x_1=aux_ind_x_1; ··· ; x_n=aux_ind_x_n;
```

provided $\{x_1, \cdots, x_n\} \cap (FV(e_1) \cup \cdots \cup FV(e_n)) \neq \emptyset$ □

Variable declarations only introduce the declared variables in scope.

Rule 11 $[\![\mathbf{var}\ x_1 : T_1; \cdots ; x_n : T_n \bullet Act]\!]^{Act} =$

```
    { (JType T_1) x_1; ··· ; (JType T_n) x_n; [[ Act ]]^Act }
```

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

Rule 12 $[\![\mathbf{if}\ g_1 \rightarrow A_1 \,\square \cdots \square\, g_n \rightarrow A_n\ \mathbf{fi}]\!]^{Act} =$

```
    if(JExp g_1){ [[ A_1 ]]^Act }··· else if(JExp g_n){ [[ A_n ]]^Act }
    else { while(true){} }
```

Finally, an assumption $\{g\}$ is implemented using an `if` block: it behaves like *Skip* if the predicate *JExp g* is true and like *Chaos* otherwise.

4.5 Compound Processes

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 13 $[\![N]\!]^{Proc}$ =
```
(new CSProcess(){
    public void run(){(new N(ExtChans ν)).run();}
  }).run();
```

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

The sequential composition of processes is easily translated. 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.

$P_1 \sqcap \cdots \sqcap P_n$ randomly chooses a process. Its definition is very similar to the corresponding one for actions.

The translation of parallelism executes a `Parallel` process that executes all the processes that are elements of the array given as argument to its constructor. Furthermore, the translation of parallelism of processes does not have to take into account variable partitions.

Rule 14 $[\![P_1 \mathbin{[\![cs]\!]} P_2]\!]^{Proc}$ =
```
(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* is actually ignored.

The renaming operation $P[x_1, \cdots, x_n := y_1, \cdots, y_n]$ is translated by replacing all the `x_is` by the corresponding `y_is` 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 15 $[\![\; [\![cs]\!] \; x_1 : T_1; \cdots; \; x_n : T_n \bullet P]\!]^{Proc}$ =
```
(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.

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 16 $[\![Types\ AxDefs\ ChanDecls\ ProcDecls\,]\!]^{Program}\ proj =$
$(DeclAxDefCls\ proj\ AxDefs)\,([\![ProcDecls\,]\!]^{ProcDecls}\ proj)$

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.

4.7 Synchronisation Channels and Indexing Operator

In this section, we extend the types of communications considered in our strategy; we deal with communication events of the form N.Expression. Our strategy implements synchronisation using array of channels. We consider the declaration of the channel *gasDischarged*, which is used throughout this section to illustrate the definitions: **channel** *gasDischarged* : *AreaId*.

By way of illustration, the synchronisation *gasDischarged*.0, which represents a gas discharge in area 0, is implemented as the 0th element in an array of channels *gasDischarged* (`gasDischarged[0]`). Basically, each synchronisation *.exp* is implemented as an additional dimension in an array of channels. In order to simplify our definitions, we consider that the use of such channels first declare possible synchronisation of the form .Exp, and finally possible communications of the form ?N or !Exp. Our strategy still constrains the channels to have only one input or output value. Multiple inputs and outputs must be encapsulated in Java objects. Thus, channels access like $c!0?x$ and $c!0.0$ are not considered in this translation strategy.

Another important constraint is that if a channel c is used in a synchronisation of the form N.Exp, it must be declared as **channel** $c : T$, where T is finite. This constraint arises from the fact that our strategy uses arrays of channels for representing synchronisation events. In order to determine the dimension of the arrays, we use the maximum and the minimum values of the type of the channel. In the case of infinite types, we would not be able to calculate the dimension of the arrays.

A very important add-on in this extension is the use of a new channel environment $\varsigma : \mathsf{N} \to SC$. It maps each channel used in the system to a value of type SC, which indicates if the channel is a communication channel (C), or a synchronisation channel (S). In our example, the channel *gasDischarged* is mapped to the value S. Basically, the changes in the translation strategy are concerned with the declaration, instantiation, and use of these channels; all the previously defined functions that are used to translate these aspects of the *Circus* programs are redefined.

First, the function *VisCDecl* is changed in order to deal with the possibility of channel array declarations. Besides the type and the name of the channel, this function, and others that follow, use an auxiliary function *ArrayDimSync* in order to check the dimension of the array of channels that implements the given channel. If this dimension is equal to zero, the channel is implemented, as in previous definitions, as a single channel.

The function *ArrayDimSync* receives three arguments: the type of the channel, *tps*, a value *sc* of type *SC*, and a *gap* that can be used to decrease the final array dimension; later on in this section, the need for this argument is explained.

If the channel is untyped, the list *tps* is a singleton with the element *Sync*. In this case, the dimension is zero. However, the dimension for typed channels is as follows. If the channel is a communication channel (C), the last type indicates the type that is communicated, and therefore, we remove one from the final array dimension. Otherwise, if no value is communicated through a channel (S), the dimension of the array is equal to the size of the *tps* list. We return as many `[]`, as the dimension we calculated for the array. We use the notation $(\texttt{code})^n$ to represent n repetitions of `code`; if $n \leq 0$, $(\texttt{code})^n$ is the empty string ϵ.

$$
\begin{array}{l}
\mathit{ArrayDimSync}\ \mathit{tps}\ \mathit{sc}\ \mathit{gap} = \\
\quad \textbf{let}\ \mathit{dim} = \begin{pmatrix} \textbf{if}\ (\mathit{tps} = [\mathit{Sync}])\ \textbf{then}\ 0 \\ \textbf{else if}\ (\mathit{sc} = C)\ \textbf{then}\ \#\mathit{tps} - 1\ \textbf{else}\ \#\mathit{tps} \end{pmatrix} \textbf{in}\ \texttt{[]}^{\mathit{dim}-\mathit{gap}}
\end{array}
$$

A local definition is used to make the definition more concise: we use the notation **let** $n = e$ **in** p to represent the substitution in p of n by e.

For channel *gasDischarged*, we have that $\mathit{tps} = [\mathit{AreaId}]$, $\mathit{sc} = S$. Besides, every calculation of an array dimension starts with a *gap* equals to zero. The application of function *ArrayDimSync* with these arguments returns the string `[]`. For this reason, in the process *FC*, the function *VisCDecl* returns the Java code `private ChannelInput[] gasDischarged`.

We also redefine the function *HidCDecl*. The definition of the dimension of possible arrays of channels is the same as for the visible channels. However, we declare the channels as `Any2OneChannel` channels. The redefinition of function *VisCArgs* is very similar to the original one. However, it also takes into account the existence of possible channel arrays, using the auxiliary function *ArrayDimSync*.

If a hidden channel is not declared as an array the channel is instantiated as a `Any2OneChannel` channel. Otherwise, we use the auxiliary function *InstArraySync* to instantiate the channel as an array of channels. The function *InstArraySync* instantiates an array of channels. It receives the types (*tps*) used in the declaration of the channel, and a value *sc* of type *SC*. If we have only one type in the list of types used in the channel declaration, we use the function *BaseCase* to declare either a channel (C) or a array of channels (S) instantiation. Otherwise, we instantiate an array of channels with dimension defined by the function *ArrayDimSync*. The function *TypeInstSync* is used to instantiate each of the elements in the array.

```
$\mathit{InstArraySync}\ \mathit{tps}\ \mathit{sc} =$
    $\textbf{let}\ \mathit{tp} = (\mathit{head}\ \mathit{tps}), \mathit{dim} = (\mathit{ArrayDimSync}\ \mathit{tps}\ \mathit{sc}\ 0)\ \textbf{in}$
        $\textbf{if}\ (\#\mathit{tps} = 1)\ \textbf{then}\ \mathit{BaseCase}\ \mathit{tp}\ \mathit{sc}\ \textbf{else}$
            new Any2OneChannel $\mathit{dim}$
            {$\mathit{TypeInstSync}\ \mathit{tps}\ \mathit{sc}\ (\mathit{Max}(\mathit{JType}(\mathit{tp}))) - (\mathit{Min}(\mathit{JType}(\mathit{tp}))) + 1$}
```

Our example falls in the first case: we have that *BaseCase AreaId S* instantiates this channel. If the channel is a communication channel (C), the function *BaseCase* instantiates a single `Any2OneChannel()` channel; otherwise (S), it uses the expression `Any2OneChannel.create(`$(\mathit{Max}(\mathit{JType}(T)))$`-`$(\mathit{Min}(\mathit{JType}(T)))$`+1)`to instantiate an array of channels with the number of elements equals to the number of possible values of the type T given as argument. The functions *Max* and *Min* return the code `MAX_T` and `MIN_T`, which represent the maximum and the minimum values in the Java type T given as argument, respectively. The channel *gasDischarged* is instantiated with `Any2OneChannel.create(MAX_AREA_ID - MIN_AREA_ID +1);`.The function *TypeInstSync* invokes the function *InstArraySync* for the remaining types of the channel declaration (*tail tps*) as many time as given as argument.

Most of the translation of actions remain the same; only those that are concerned with communications and external choice must be extended. For instance, for a given input channel c, the type of the communicated value is given by the Java type of the last element in the list (*last*) of types of c. This is the type used to declare the new declared variable. Each synchronisation $.i$ is translated to an access of the i-th element in an array of channels.

Rule 17 $[\![c\ .e_0\ \cdots\ .e_m?x \rightarrow Act]\!]^{Act} =$
$\quad$ ***let*** $commType = JType(last\ \ (snd\,(\delta\ \ c)))$ ***in***
$\quad$ `{`$commType$ `x=(`$commType$`)c[`$JExp\ e_0$`]` $\cdots$ `[`$JExp\ e_n$`].read();` $[\![\,Act\,]\!]^{Act}$ `}`

Given a triple (a, b, c), the functions *fst*, *snd*, and *trd* return a, b, and c, respectively. The translation of output and synchronisation channels is simply changed in the same way; we omit the rule definitions for conciseness.

In the case of external choice, we must redefine some of the auxiliary functions (*ICAtt*,*ExIC*,*DeclCs*,*Cases*) to take into account the existence of arrays of channels; each channel in an array of channels is considered as a different initial channel. For instance, the function *DeclCs*, used to declare one constant for each channel that takes part in the external choice, returns, for each synchronisation $c.x_0 \cdots .x_m$, a declaration of a constant `CONST_C_X_0_`$\cdots$`X_m`.

For example, in the choice $a.0 \rightarrow \cdots \,\Box\, a.1 \rightarrow \cdots$, we consider $a[0]$ and $a[1]$ as initial channels; this is reflected in the declaration of the guards, the constants, and the `switch` block, as presented below.

```
Guard[] guards = new Guard[]{a[0],a[1]};
final Alternative alt = Alternative(guards);
final int CONST_A_0 = 0; final int CONST_A_1 = 1;
switch(alt.select(g)){case CONST_A_0: { a[0].read(); ... } break;
                      case CONST_A_1: { a[1].read(); ... } break;}
```

This concludes the translation of our example.

An indexed process can be seen as a kind of parameterised process. The difference, however, is that a syntactic substitution on the channels is made. It is very important to notice that the creation of the channels environment already takes into account the indexed processes. So, the channels implicitly created by the indexed operator are already within the channel environment. For instance, consider the following channel environment $\delta \mathrel{\widehat=} \{c \mapsto ([], [\mathbb{N}])\}$ that has a channel c of type $\mathbb{N}$. In the translation of the process $i : T_1 \odot Proc$, we consider that the environment δ is changed to $\delta \mathrel{\widehat=} \{c\_i \mapsto ([], [T_1, \mathbb{N}])\}$.

The renaming in the channels within a given indexed process $Decl \odot Proc$ is reflected in the way the channels are instantiated, referenced, and used. An indexed process $x_1 : T_1;\ \cdots;\ x_n : T_n \odot Proc$ is translated as a parametrised process $x_1 : T_1;\ \cdots;\ x_n : T_n \bullet Proc$ but, for every channel c used within the process, we replace every reference to c, by a reference to $c\_x\_1 \cdots x\_n.x_1.\ \cdots\ .x_n$.

Rule 18 $[\![\,x_1 : T_1;\ \cdots;\ x_n : T_n \odot Proc\,]\!]^{ParProc}\ P =$
$\quad [\![\,(x_1 : T_1;\ \cdots;\ x_n : T_n \bullet Proc)$
$\qquad [c : used(Proc) \bullet c\_x\_1 \cdots x\_n.x_1.\ \cdots\ .x_n]\,]\!]^{ParProc}\ P$

The process $P[c : \mathit{used}(\mathit{Proc}) \bullet c\_x\_1 \cdots x\_n.x_1. \;\cdots\; .x_n]$ is that obtained from P by changing all the references to a used channel c by a reference to the channel $c\_x\_1 \cdots x\_n$, with synchronisation $x_1. \;\cdots\; .x_n$.

An instantiation of an indexed process is translated as an invocation of a parameterised process. However, the same syntactic substitution as the one present in the rule above is made before the translation.

Rule 19 $[\![(x_1 : T_1; \;\cdots;\; x_n : T_n \odot \mathit{Proc})\lfloor v_1, \cdots, v_n \rfloor]\!]^{\mathit{Proc}} =$
$[\![((x_1 : T_1; \;\cdots;\; x_n : T_n \bullet \mathit{Proc})$
$[c : \mathit{used}(\mathit{Proc}) \bullet c\_x\_1 \cdots x\_n.x_1. \;\cdots\; .x_n])\,(v_1, \cdots, v_n)]\!]^{\mathit{Proc}}$

If the instantiation uses the process name $N\lfloor v_1, \cdots, v_n \rfloor$, we simply translate it as $[\![N(v_1, \cdots, v_n)]\!]^{\mathit{Proc}}$. As the process N is defined as an indexed process, its translation takes into account the new channels created by the indexing operator.

The iterated indexed sequential composition over the indexed operator can be simply translated as an iterated sequential composition. However, the process has all the references to the channels within it changed before the translation.

Rule 20 $[\![(\fatsemi\, x_1 : T_1; \;\cdots x_n : T_n \odot \mathit{Proc})]\!]^{\mathit{Proc}} =$
$[\![(\fatsemi\, x_1 : T_1; \;\cdots x_n : T_n \bullet$
$(\mathit{Proc}[c : \mathit{used}(\mathit{Proc}) \bullet c\_x\_1 \cdots x\_n.x_1. \;\cdots\; .x_n]))]\!]^{\mathit{Proc}}$

For the same reasons as those for iterated external choices, the iterated indexed external choice must be expanded before being translated. The iterated indexed internal choice can also be simply translated as an iterated internal choice with the process having all the references to the channels within it changed before the translation. Its definition is very similar to that of the iterated indexed sequential composition. The same strategy applies to the translation of iterated indexed parallelism and interleaving.

4.8 Generic Channels

In this section, we deal with generic channels. We consider the declaration of the channel *lamp* used in our case study, which is used throughout this section to illustrate the definitions: **channel**$[T]$ *lamp* $: T \times \mathit{OnOff}$. This declaration introduces a family of channels *lamp*, used in our case study by the system to switch a given lamp on or off; T is used as a parameter to determine the type of the lamp that is used in the communication of a value of type *OnOff*. The syntax used for synchronisation channels still holds throughout this section; the only difference from Section 4.7 is the possibility of generic channel instantiation.

Basically, each generic typing variable in a generic channel declaration is implemented as an additional dimension in an array of channels: each element represents a possible instance of the channel. For example, the instantiation of the generic channel *lamp*$[\mathit{AreaId}]$ from our example, is implemented as `lamp[Type.AREA_ID]`.

Our translation strategy assumes that every type used within the system is already implemented in Java. Besides, these must inherit from the class `Type`, which has an integer constant for each type used within the system. The translation strategy translates references to a type into a reference to the corresponding constant in class `Type`, as in the example presented above where the reference to type *AreaId* is translated to `Type.AREA_ID`.

A generic process declaration is translated as a process parameterised by the types used in the declaration. For this reason, if we have a generic process P, we consider the type arguments as arguments of P in its translation, and replace every reference to that type identifier, by the primitive value of the integer given as argument. Besides, any reference to the generic type variable is replaced by a reference to the superclass `Type`. The typing variables are not types; we assume that *JType* returns the names given as arguments in these cases.

Rule 21 $[\![\textbf{process}\ P[T_0, \cdots, T_n] \mathrel{\widehat=} Decl \bullet Proc\,]\!]^{Proc}\ proj =$
$\quad ReplTRefs\,([\![\textbf{process}\ P \mathrel{\widehat=} t_0 : \mathbb{N};\ \cdots;\ t_n : \mathbb{N};\ Decl \bullet Proc\,]\!]^{Proc}\ proj)$

Given a Java code c and for every type T used to instantiate a generic process, the function *ReplTpRefs* replaces every occurrence of *JType* T in c by `Type`, and every occurrence of `Type.`$CJType(T)$ in c by `t.intValue()`. For a given type T, the function *CJType*, returns the name of the Java type of T with all the letters capitalised. This is far from essential, but is kept in the translation in order to follow the Java coding standards.

Instead of using the previously defined function *ArrayDimSync*, the functions *VisCDecl*, *HidCDecl*, *VisCArgs*, and the instantiation of hidden channels, use the function *ArrayDim* defined below in order to find out the dimension of the possible array of channels. It receives four arguments: a list *genTypes* of the generic type arguments of the channel, and the three arguments used by the function *ArrayDimSync*. This function adds the number of generic types used in the declaration of a given channel to the result of *ArrayDimSync*.

```
$ArrayDim\ genTypes\ tps\ sc\ gap =$
    $\textbf{let}\ \ dim = \#genTypes + (ArrayDimSync\ tps\ sc\ gap)\ \textbf{in}\ \texttt{[]}^{\,dim-gap}$
```

For channel *lamp*, we have $genTypes = [T]$, $tps = [T, OnOff]$, $sc = C$, and $gap = 0$. The application of function *ArrayDim* with these arguments returns the string `[][]`. The array dimension for this channel is two, one for being a single generic channel and another for being a synchronisation channel; *VisCDecl* returns the Java code `private ChannelOutput[][] lamp`.

As in Section 4.7, the changes in the translation are in the declaration, instantiation, and use of the generic channels. In the channel declaration and instantiation, the only difference from Section 4.7 is that we replace the use of *InstArraySync* by the use of *InstArray*. First, it deals with the dimensions related with the generic variables, and then it uses the previously defined function *InstArraySync*, in order to deal with dimensions that are originated from synchronisation. It receives the generic type arguments (*genTypes*) and the types (*tps*) used in the declaration of the channel, a value *sc* of type *SC* indicating if the channel is a synchronisation or a communication channel, an a list of all types used within the system (τ).

For each generic type variable, this function instantiates an array of channels with a dimension determined by the function *ArrayDim*. This instantiation uses an auxiliary function *GenericInst*, which declares an element in the array for each type T_n used within the system (τ). Finally, when all the generic type variables have been dealt with, the function invokes the function *InstArraySync*, in order to instantiate any further arrays related to possible synchronisation.

```
$InstArray\ genTypes\ tps\ sc\ \tau =$
    $\textbf{let}\ dim = (ArrayDim\ genTypes\ tps\ sc\ 0)\ \textbf{in}$
        $\textbf{if}\ (\#genTypes > 0)\ \textbf{then}$
            new Any2OneChannel $dim$\{ $GenericInst\ genTypes\ tps\ sc\ \tau\ \tau$ \}
        $\textbf{else}\ InstArraySync\ tps\ sc$
```

The expression $InstArray\ (tail\ genTypes)\ (replace(head\ genTypes, T, tps))\ sc\ \tau$ is recursively invoked by function *GenericInst*, for each type T in τ, in order to deal with the remaining (if any) generic type variables. Each of these invocations corresponds to an instantiation of the first generic variable in the list with T; for this reason, we replace in *tps* every reference to the first variable in the list (*head genTypes*) by the type T using the function *replace*.

Our example falls in the first case of function *InstArray*. As previously discussed, for this channel, we have $genTypes = [T]$, and so $\#[T] = 1 > 0$. As we already know the array dimension for this channel is two, so we have the following instantiation for channel *lamp*: `new Any2OneChannel[][]{`$GenericInst\ [T]\ [T, OnOff]\ C\ \tau\ \tau$ `};`. One of the eight types used in our case study was the one related to the identification of the areas, *AreaId*. By way of illustration, part of code generated by *GenericInst* is $InstArray\ (tail\,[T])\ [AreaId, OnOff]\ C\ \tau \cdots$. Each of these lines corresponds to a certain type within the system. Notice that the function *replace* has replaced the variable T, in the types list $[T, OnOff]$ by the corresponding type (*AreaId* in our example). Besides, we have that $tail\,[T] = [\,]$. For this reason, the invocation of function for each of the eight elements above, we have that the function *InstArray* above yields $InstArraySync\ [AreaId, OnOff]\ C$. The same happens to the remaining types of the system. By applying the definition of *InstArraySync*, we notice that the dimension of the array is now one. Hence, we have the following: `new Any2OneChannel[]{` $TypeInstSync\,[AreaId, OnOff]\,C\,2$ `}`. The result of the function *TypeInstSync* contains two comma-separated invocations to the function *InstArraySync*: $InstArraySync\,[OnOff]\,C$ `,` $InstArraySync\,[OnOff]\,C$. Again, following the definition of *InstArraySync*, each invocation leads to the base case, which, since we have a communication channel (C), returns a single channel instantiation `new Any2OneChannel()`. This concludes the instantiation of the channel *lamp*. A illustration of the instantiation of this channel is presented below; its full instantiation can be found in [14].

```
this.switchLamp = new Any2OneChannel[][]{
                 // Type AreaId
                  new Any2OneChannel[]{new Any2OneChannel(),
                                       new Any2OneChannel()},...};
```

We are left now with the usage of these channels.

As for synchronisation channels, just a few rules must be redefined. These redefinitions are straightforward. Basically, we extend the definitions for synchronisation channels, by taking into account possible generic channel instantiations. For a given input or output channel c, the type of communicated value (*commType*) is given by the Java type of the last element in the list (*last*) of types of c. This is the type used to declare the new variable.

Rule 22 $[\![c\ [T_0, \cdots, T_n].e_0 \cdots .e_m?x \rightarrow Act]\!]^{Act} =$

$\quad$ ***let*** $commType = JType(last\ (snd\,(\delta\ c)))$ ***in***

$\qquad$ `{` $commType$ `x = (`$commType$`)c[Type.(`$CJType\ T_0$`)]` $\cdots$ `[Type.(`$CJType\ T_n$`)]` `[`$JExp\ e_0$`]` $\cdots$ `[`$JExp\ e_n$`].read();`

$\qquad$ $[\![Act\,]\!]^{Act}$ `}`

The translations of output and synchronisation channels is simply changed in the same way and are omitted.

As for synchronisation channels, in the case of external choice, we must redefine some of the auxiliary functions to take into account the existence of arrays of channels; each channel in an array of channels is considered as a different visible channel. These redefinitions are pretty straightforward. For instance, the function *ICAtt*, which is used in the declaration of the array of channels that take part in the external choice, takes into account possible generic channels and synchronisation values in the channels. The translation of channels in the form $c.[T_0, \cdots, T_n]x_0 \cdots .x_m$ yield `c[Type.(`$CJType\ T_0$`)]`$\cdots$`[Type.(`$CJType\ T_n$`)] [`$JExp\ x_0$`]`$\cdots$`[`$JExp\ x_m$`]`.

For each channel in the form presented above that take part in a external choice, the function *DeclCs* yields the declaration of a constant as follows.

```
final int CONST_C_($CJType(T_0)$)_ $\cdots$ _($CJType(T_n)$)_X_0_ $\cdots$ X_m = n
```

Finally, the redefined function *Cases*, which returns a sequence of Java `case` blocks, one for each initial channel in given channel list, returns a different case block for each element in an array of channel that takes part in the external choice.

As discussed before, the types are declared as arguments of a generic process. For this reason, we must use the constants which represent each of the types used in the instantiation. These are given as Java `Integer`s, which are constructed using the corresponding constants, to the constructor of the class corresponding to the process that is being instantiated.

Rule 23 $[\![N[T_0, \cdots, T_n](e_0, \cdots e_n)]\!]^{Proc} =$

```
(new CSProcess(){
    public void run() {
        (new N(new Integer(Type.($CJType\ T_0$)),$\cdots$,
               new Integer(Type.($CJType\ T_n$)),
               ($JExp\ e_0$), $\cdots$ ($JExp\ e_n$), $ExtChans\ \nu$)).run(); }
}).run();
```

For parametrised processes, we have that *JExp e* is also given to the constructor, for each parameter e, used to instantiate the process. Finally, as expected, a ,-separated list of visible channels of the process is also used to instantiate the process.

4.9 Multi-synchronisation

In this section, we deal with multi-synchronisation channels. First, we present some Java components that were implemented by us for use in this translation. Then, we present the translation rules.

We implement multi-synchronisation using a centralised solution based on the work presented in [23]: the distribution of a multi-synchronisation is replaced by a process that controls the multi-synchronisation in a given channel, and by client processes that potentially synchronise on the channel.

Two components are used: a process that represents the synchronisation controller (`MultiSyncControl`) and a process that represents the synchronisation client (`MultiSyncClient`). For each channel involved in a multi-synchronisation, we have a controller; each time a process is willing to engage in a multi-synchronisation, we must instantiate a new client process and run it. At the end of the execution, possibly communicated values can be retrieved from the client.

In Figure 4, we illustrate an architecture using these components for two channels involved in a multi-synchronisation and four processes. In this example, we have one *MultiSyncControl* for each channel, and each process instantiates its own *MultiSyncClient*. The controllers use an array of channels *fromSync* to communicate with each of their clients. The clients share a channel *toSync* to communicate with their controller. This channel is not multi-synchronised since is JCSP communications happens only between two processes only.

The three top-most clients synchronise on the channel that is controlled by right-hand side controller, and the three bottom clients synchronise in the channel controlled by the left-hand side controller. Each of the clients have a different identification regarding each of the controllers. For instance, the second client from the top, is identified as client zero on the left, and as client 1 on the right. In this paper, we extend the work presented [23]: clients may take part in more than one

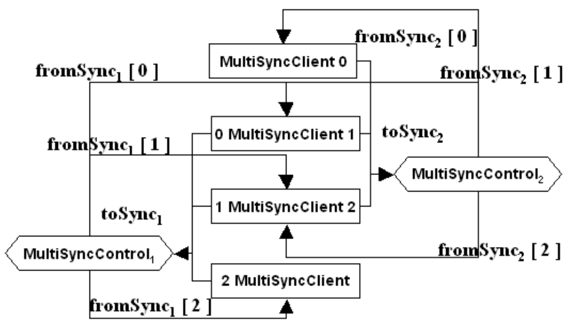

Fig. 4. Architecture for the Multi-synchronisation components

multi-synchronisation, in non-multi-synchronised communications, and values may be carried through channel involved in a multi-synchronisation.

Another environment is considered to be available throughout the translation strategy: the environment $\omega : \mathsf{N} \nrightarrow ((\mathsf{N} \rightarrowtail\!\!\!\!\nrightarrow \mathbb{N}) \times \mathbb{N} \times \mathsf{N})$ includes a triple for every name of channel involved in a multi-synchronisation. The first element is function that gives an identification number for every process involved in the multi-synchronisation. Our strategy considers that these identifications start from zero, and are incremented by one for each process. For instance, consider a channel c, in which processes P_0, P_1, and P_2 synchronise. A possible identification function would be $ID = \{P_0 \mapsto 0, P_1 \mapsto 1, P_2 \mapsto 2, \}$. The second element in the triple is the number of processes that are involved in the multi-synchronisation. This can be easily calculated from the cardinality of the domain of the identification function, but we keep it for conciseness in the definitions. The third, and last, element in the triple is the name of the process that is writing to the channel; following a limitation from JCSP, all other processes are considered readers.

Consider a system with only one channel c used for multi-synchronisation and three process P_0, P_1, and P_2, the writer, that synchronise on c. In this case, the environment ω could be defined as $\{c \mapsto \{ID, 3, P_2\}\}$, where ID is as above.

In order to create a `MultiSyncControl` process that controls the synchronisation of channel c, the user must give the array of channels `from_c`, and the channel `to_c` that the clients use to communicate with the controller, as arguments. The number of clients can be easily retrieved from the ω environment $(\mathit{JExp}(\mathit{snd}(\omega\ c)))$.

```
Any2OneChannel[] from_c = Any2OneChannel.create(3);
Any2OneChannel to_c = new Any2OneChannel();
MultiSyncControl c = new MultiSyncControl(from_c, to_c);
```

The instantiation of the clients requires a little bit more of information. For each multi-synchronisation c on which the process takes part we must create a synchronisation object that contains the channel used by this client in the array of channels `from_c`, the channel `to_c`, the identification of the process in this multi-synchronisation $(\mathit{JExp}((\mathit{fst}(\omega\, c))\, P) = 0)$, and the identification of the writer in that channel $(\mathit{JExp}((\mathit{fst}(\omega\, c))\, (\mathit{trd}(\omega\, c))) = 2)$. For instance, in the translation of process P_0, we create the following synchronisation object for channel c.

```
Object[] sync = new Object[]{from_c[0], to_c, 0, 2};
```

Next, we create a `Vector` that contains all these multi-synchronisation objects.

```
Vector sqOfSyn = new Vector(); sqOfSyn.addElement(sync);...
```

A `Vector` of channels that are not involved in a multi-synchronisation is also used in the instantiation of a client.

For instance, consider we have a channel nm, which is not multi-synchronised. We have the following Java code.

```
Vector sqOfNSyn = new Vector(); sqOfSyn.addElement(nm);
```

Finally, we instantiate the client and execute it as follows.

```
MultiSyncClient client = new MultiSyncClient(sqOfSyn,sqOfNSyn,v);
client.run();
```

The last argument v is the value communicated through the channel. If this client is the writer, this is the value that will be communicated to the readers once the synchronisation happens. If this process is not the writer, we have that `v=null`.

Most of the extension for dealing with multi-synchronisation is done simply by replacing code in the Java code generated by the translation of the previous rules. These replacements change only those parts of the Java code that are related to the multi-synchronisation channels. Basically, every reference to a channel involved in a multi-synchronisation is replaced by a reference to the channels use in the communication with the multi-synchronisation controller.

Furthermore, the translation of external choice is changed in order to deal with channels involved in a multi-synchronisation, and the translation of the body of a process is also changed in order to include the execution of the controllers for the channels involved in a multi-synchronisation that are hidden within that process. All the changes are discussed later in this section.

The first code substitution deals with the declaration of every hidden channel c involved in a multi-synchronisation within a process. We replace every declaration of such channel by the declaration of a channel `private Any2OneChannel to_c` and the channel array `private Any2OneChannel[] from_c` used to communicate with the multi-synchronisation controller.

In a similar way, we also make a substitution of the declarations of visible channels involved in a multi-synchronisation. However, since the control of reading and writing in a channel involved in a multi-synchronisation is now left to the synchronisation controller, we do not make any distinction between input and output channels. The same applies in the declaration of the class constructors arguments that are related to the channels involved in a multi-synchronisation.

Next, we replace instantiations of these channels by instantiations of the channels that make the communications between servers and clients as follows. The environment ω gives how many processes take part in the multi-synchronisation.

```
this.from_c = Any2OneChannel.create(JExp(fst(ω c)));
this.to_c = new Any2OneChannel();
```

The initialisation of visible channels in the constructor is also replaced by the assignment of both channels related to c:

```
this.from_c = newFrom_c; this.to_c = newTo_c;
```

Every access to a channel involved in a multi-synchronisation must be made in the ways explained before. Each time that a multi-synchronisation is used, we must instantiate a client and execute it. Given an index ind, the function *InstMultiSync* returns the Java code that implements a multi-synchronisation on channel c within a process P. The index of the right channel within the array `from_c` is the result of the expression $\mathit{JExp}((\mathit{fst}(\omega\ c))\ P)$, which is the Java expression corresponding to the identity of the process in the synchronisation. The identity of the writer is given as the application of the identity function to name of the writer process, thus $\mathit{JExp}((\mathit{fst}(\omega\ c))\ (\mathit{trd}(\omega\ c)))$.

We present below how a multi-synchronisation within P_0 must be implemented.

```
Vector sqOfSyn_0 = new Vector();
Object[] sync_0 = new Object[]{from_c[0],to_c,0,2};
sqOfSyn.addElement(sync_0);
MultiSyncClient client_0 =
   new MultiSyncClient(sqOfSyn_0, new Vector(),null);
client_0.run();
```

First, we replace any channel reading that stores the read value, and the we replace the remaining communications. The reading $T\,x$`=(`T`)c.read()` from every multi-synchronised channel c, where T is any java type, is replaced by the following: we execute a client, as the one presented above, and finally, we retrieve the communicated value from the client $T\,x$`=(`T`)client_`*ind*`.getValueTrans();`. Writing (`c.write(x)`) to a channel and reading (`c.read()`) from a channel (if we do not store the read value) are also replaced by a client execution. Processes that write a value x are considered to be the channel writer; therefore, in this case, we also replace the `null` value used in the instantiation of the client, by x.

The only rules that need changes are those for parameterised processes and for external choice. The new translation of external choice is very similar to the one presented before. The only change from Rule 6, is that it does not use the `Alternative` JCSP class, but a multi-synchronisation client. For every multi-synchronised channel c involved in the external choice, we create a synchronisation object corresponding to c and insert it the vector `sqOfSync_`*ind*; the remaining channels are inserted in the vector `sqOfNSync_`*ind*. Finally, as explained before, we create a multi-synchronisation client `client_`*ind* and execute it. At the end of its execution, `client_`*ind* has the index of the chosen channel, and possibly a communicated value. Since there is no guarded output, every channel that takes part in an external choice is not willing to write anything to the channel, hence, the communicated value used to instantiate the *MultiSyncClient* is `null`. Finally, the translation of `switch` block remains almost the same: the choice is actually retrieved from the client using the method `getChosen`, which returns the index of the chosen channel.

In the previous translation of external choice, after choosing a channel in the `switch` block, the first line of the code is the translation of the channel reading. However, in the case of multi-synchronisation, this is already done by the multi-synchronisation client. For this reason, the translation of the `case` bodies related to the channels involved in a multi-synchronisation are slightly changed: we replace `(`T`)`x `= (`T`)c.read()` by `(`T`)`x `=(`T`)client_`*ind*`.getValueTrans()` and remove the remaining readings `c.read();`.

Every channel is instantiated within the process that hides it; otherwise it is received and initialised in the constructor. For this reason, we chose to instantiate the multi-synchronisation controller for a given channel in the same class in which c is instantiated. First, we need to retrieve the information of which multi-synchronisation channels are hidden within a process. This can be easily defined as the intersection of the hidden channels of a process (dom ι) and the multi-synchronisation channels (dom ω). We emphasise that the environment ι stores information for every hidden channel in a certain process. For a given process whose multi-synchronisation channels are $c_1,\cdots,c_n$, we have that the process that represents the parallelism of controllers of $c_1,\cdots,c_n$ is defined as follows.

$$\begin{aligned} \mathit{Controllers} \mathrel{\widehat{=}}\ & \mathit{MultiSyncControl}(\mathit{from\_c}_1, \mathit{to\_c}_1) \\ & \parallel \cdots \parallel \mathit{MultiSyncControl}(\mathit{from\_c}_n, \mathit{to\_c}_n) \end{aligned}$$

We redefine Rule 2 for parametrised processes, by changing the translation of the `run` method body. Instead of simply translating the process definition, we translate the parallel composition of the process body with the controllers. Furthermore, we use the function *ReplMultiSync*, in order to apply all the substitutions previously discussed to the Java code related to the channels involved in a multi-synchronisation.

`public void run(){` $\mathit{ReplMultiSync}\ (\operatorname{dom}\omega)\ \ [\![\, \mathit{Proc} \parallel \mathit{Controllers}\,]\!]^{\mathit{Proc}}$ `}`

For every channel c in a set of channels s, the application $\mathit{ReplMultiSync}\ s\ jc$, where jc is a Java code, applies the substitution described in this section to jc.

One last rule that must be changed is the rule that translates process invocations. This is needed to deal with the fact that we actually have invocations of a very special process (*MultiSyncControl*), which needs a slightly different translation strategy: we do not need to give each of the external channels to this process. We use only two arguments: the array of channels used by the controller to communicate with the clients, and the channel used to communicate with the controller.

Rule 24 $[\![\, \mathit{MultiSyncControl}(\mathit{from\_c}, \mathit{to\_c})\,]\!]^{\mathit{Proc}} =$

```
(new CSProcess(){
public void run(){(new MultiSyncControl(from_c,to_c)).run();}
 }).run();
```

This extension of the translation strategy presented in [15] was vital in the implementation of our case study since multi-synchronisation plays a major role in our system. The implementation of the whole system can be found in [14]. For conciseness, we omit the translation of free types and abbreviations [14].

5 Case Study

Our case study [16] consists of a fire control system that covers two separate areas. Each area is divided into two zones; two different zones cannot be covered by two different areas. Fire detection happens in a zone and a gas discharge may occur in the area that contains that zone. The system includes a display panel composed of lamps that indicates whether the system is on or off, whether there are system faults, or a fire has been detected, whether the alarm has been silenced or not, the need to replace the actuators of the system, and gas discharges.

The system can be in one of three modes: manual, automatic, or disabled. In manual mode, an alarm sounds when a fire is detected, and the corresponding detection lamp is lit on the display. The alarm can be silenced, and, when the reset button is pressed, the system returns to normal. In manual mode, gas discharge is manually initiated. In automatic mode, a fire detection is followed by the alarm being sounded; however, if a fire is detected in the second zone of the same area, the second stage alarm is sounded, and a countdown starts. When it finishes, the gas is discharged and the circuit fault lamp is lit on the display; the system mode is switched to disabled. In disabled mode, the system can only have the actuators replaced, identify relevant faults in the system, and be reset. The system is back to its normal mode after the actuators are replaced and the reset button is pressed.

The motivation for the fire control system refinement is the distribution of the areas, in order to increase efficiency. In [16], we discuss in details the refinement steps summarised graphically in Figure 5.

In the first iteration, we split the abstract fire control *AbstractFC* into two process *Areas* and *InternalFC*. The first models the areas of the system, and is split into two interleaved *Area* processes in the last iteration. The second is the core of the system, which is split into a display controller *DisplayC* and the system controller *FC* in the second iteration. After the last iteration, additional simple schema

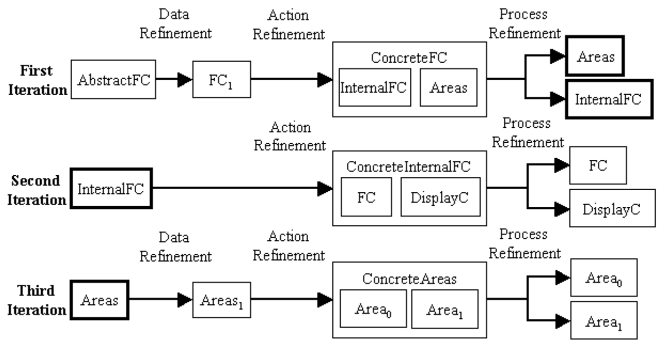

Fig. 5. Refinement Strategy for the Fire Control System

refinements using [5] were needed; this refinement yield an executable specification of the system, whose translation to Java used the strategy presented here and is discussed in the next section.

6 Implementing the Fire Control System

After translation, the classes that implement the processes are located in the package `processes`. Figure 6 presents a UML class diagram of this package after the translation strategy was applied to our case study [16]. We highlight the core of the system which was presented in this paper, the process `ConcreteFC`. This process hides multi-synchronisation channels, and for this reason, it is the one responsible for instantiating the multi-synchronisation controllers for each of these channels. On the other hand, the process that implements each of the areas in the fire control system (`Area`), the process that implements the display controller (`DisplayC`), and the process that implements the core of the fire control (`FC`) take part in multi-synchronisation, and hence, instantiate multi-synchronisation clients.

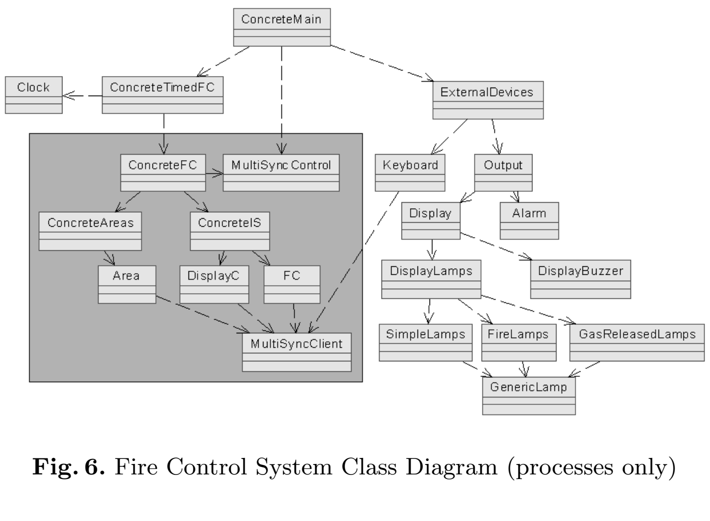

Fig. 6. Fire Control System Class Diagram (processes only)

In order to run the whole system, we have created a parallel composition of the `ConcreteFC` with a process that represents a `Clock`. The external devices where also implemented. A `Keyboard` may be used to input signal to the system. The `Output` process encapsulates all the output processes which are the `Alarm` and the `Display`.

The last is composed by a buzzer and by the lamps. There are three different instantiations of the lamps: the `FireLamps` indicate where a fire has been detected; the `GasReleasedLams` indicate where gas has been discharged; and the remaining lamps are implemented within process `SimpleLamps`. All the lamps are instantiation of the generic process `GenericLamp`. The parallel composition of the `ConcreteTimedFC` with the `ExternalDevices` represents the whole system, `ConcreteMain`.

The implementation has also included typing classes, utilities classes, which are also part of our translation strategy (*e.g.,* `RandomGenerator`), and graphic interface classes, which, although not arising from the translation, where implemented in order to allow us to interact with the system.

In Figure 7, we present a snapshot of the execution of the process `ConcreteMain`. This interface contains the following elements: gas lamps for areas 0 and 1, fire lamps for zones 0 to 5, one fault lamp for each possible fault within the system, the lamp that indicates that system is on, an alarm in the form of a progress bar (an empty bar indicates that the alarm is off, an half filled bar indicates a first stage alarm, and a full filled bar indicates a second stage alarm), a clock that shows if the clock is counting down, and a keyboard that can be used by the user to simulate inputs to the system. Besides, a sound is also played if the display buzzer is switched on.

In this snapshot, we have that fire has been detected in zones 0, 1, and 2, and that three faults have been detected: secondline fault, power fault, and isolate remote signal. In this example, the system is running in automatic mode. As specified,

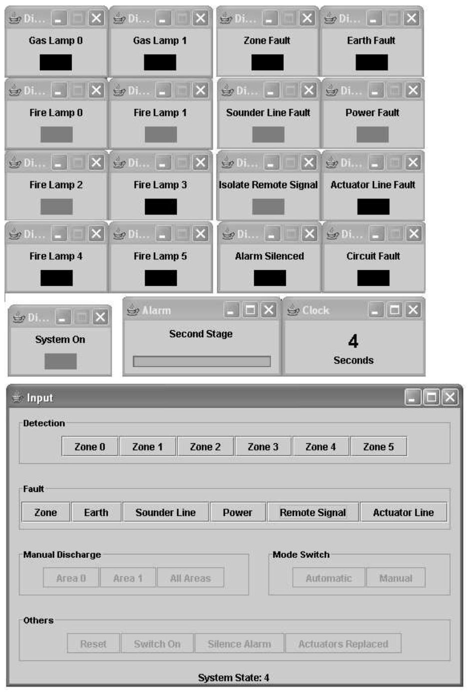

Fig. 7. Fire Control System Graphic Interface

the fact that fire has been detected in two zones in the same area, has started the counting down of the clock and has set the alarm to its second stage. After the conclusion of the counting down, the gas is charged in area 0 and this is indicated in the display by switching the gas lamp 0 on.

The implementation of the fire control system using the translation strategy yielded to 5400 lines of Java code [14]. Most of this implementation could have been mechanised if tool support, which is already under development, were already available. Throughout the translation of the case study, we could verify the correctness of our translation strategy, and even simplify the definitions of some rules. Furthermore, the translation of the case study also provided us with a industrial scale example of application of the strategy.

7 Conclusions

The translation strategy presented in this work extends the one presented in [16], by including synchronisation and generic channels, indexing operators, generic processes, and multi-synchronisation. It has been used to implement several programs, including a quite complex fire control system developed from its abstract centralised specification [14], which is also presented here. 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.

In [7], Fischer formalises a translation from CSP-OZ to annotations of Java programs. A CSP-OZ specification is composed mostly of 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 [3], where 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 [8], 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.

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.

Due to JCSP limitations, we consider a restricted set of communications: untyped inputs, outputs, and synchronisations. In this paper, we treat generic channels and synchronisations $c.e$ over a channel c with expression e. Strategies to refine out the remaining forms of communication 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 [23].

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. Furthermore, we left the investigation into the translation of nested parametrised and indexing processes is left as future work.

An 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 [16] and on the fairly direct correspondence of JCSP and *Circus*.

References

1. J.-R. Abrial. *The B-book: assigning programs to meanings.* Cambridge University Press, 1996.
2. E. Börger and R. F. Stärk. *Abstract State Machines—A Method for High-Level System Design and Analysis.* Springer-Verlag, 2003.
3. 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`.
4. A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. A refinement strategy for circus. *Formal Aspects of Computing*, 15(2-3):146–181, 2003.
5. A. L. C. Cavalcanti and J. C. P. Woodcock. Zrc - a refinement calculus for z. *Formal Aspects of Computing*, 10(3):267 – 289, 1999.
6. 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.
7. C. Fischer. *Combination and Implementation of Processes and Data: from CSP-OZ to Java.* PhD thesis, Fachbereich Informatik, Universität Oldenburg, Oldenburg - Germany, 2000.
8. G. Hilderink, J. Broenink, W. Vervoort, and A. Bakkers. Communicating java threads. In *Parallel Programming and Java Conference*, 1997.
9. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.
10. C. A. R. Hoare and J. He. *Unifying Theories of Programming.* Prentice-Hall, 1998.
11. C. B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall International, 1986.
12. R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.
13. Carroll Morgan. *Programming from Specifications.* Prentice-Hall, 1994.
14. M. V. M. Oliveira. *A Refinement Calculus for Circus - PhD Thesis Additional Material*, Dez 2005. http://www.cs.york.ac.uk/~marcel/phd/.
15. M. V. M. Oliveira and A. L. C. Cavalcanti. From circus to jcsp. In J. Davies et al., editor, *Sixth International Conference on Formal Engineering Methods*, volume 3308 of *LNCS*, pages 320 – 340. Springer-Verlag, November 2004.
16. 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.
17. P.H.Welch, G.S.Stiles, G.H.Hilderink, and A.P.Bakkers. Csp forjava:multithreading for a ll.

18. A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall Series in Computer Science. Prentice-Hall, 1998.
19. 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.
20. A. C. A. Sampaio, J. C. P. Woodcock, and A. L. C. Cavalcanti. Refinement in circus. In L Eriksson and PA Lindsay, editors, *FME 2002: Formal Methods - Getting IT Right*, volume 2391 of *LNCS*, pages 451–470. Springer-Verlag, unknown 2002.
21. J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall, 2nd edition, 1992.
22. 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.
23. 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.
24. J. C. P. Woodcock and J. Davies. *Using Z – Specification, Refinement, and Proof.* Prentice-Hall, 1996.