Automatic Translation from *Circus* to Java

Angela Freitas and Ana Cavalcanti

Department of Computer Science
University of York, UK

Abstract. *Circus* is a combination of Z and CSP that supports the development of state-rich reactive systems based on refinement. In this paper we present *JCircus*, a tool that automatically translates *Circus* programs into Java, for the purpose of animation and simulation. It is based on a translation strategy that uses the JCSP library to implement some of the CSP constructs of *Circus*. The tool generates a simple graphical interface; we present a simple example to demonstrate the translation strategy, and the execution of the resulting program. We discuss the class `GeneralChannel`, which we designed to support the implementation of multi-synchronisation. We also discuss our improvements to the translation strategy, some limitations of the tool, and our approach to prove the correctness of the multi-synchronisation protocol.

1 Introduction

Circus [1] is a combination of the Z notation [2], the process algebra CSP [3], and Djikstra's language of guarded commands. It is a unified language for specification and design of state-rich reactive systems. In general terms, data requirements are expressed with Z schemas, and behavioural aspects are expressed using Z, CSP and the guarded commands. *Circus* also includes a refinement calculus, which allows stepwise development of programs. The semantics of *Circus* is based on the Unifying Theories of Programming [4], a relational model that unifies programming theories across many different paradigms.

Circus supports specifications of systems at various levels of abstraction. In [5], a complete development strategy supported by *Circus* is presented. Starting from an abstract *Circus* specification, refinement laws are gradually applied in order to reach a concurrent implementation in which all schemas used to describe operations are refined to commands and CSP actions. Afterwards, a translation strategy is applied to generate a Java implementation. The JCSP library [6] is used for implementation of some CSP primitives in Java.

Unlike the refinement calculus, which requires human expertise to be applied, the translation strategy can be automated. Tool support is important to save effort and avoid human errors that are typical of the activity of writing code.

This paper describes *JCircus*, a tool that implements the translation strategy from *Circus* to Java. It receives as input a *Circus* program written in LaTeX

markup based on that adopted for the Z Standard [7], and produces a Java program that implements the program. *JCircus* translates concrete *Circus* programs, that is, those in which specification statements and Z schemas are not used in action definitions. The strategy covers a large subset of concrete *Circus*, including generic processes and some CSP replicated operators. Due to limitations in the JCSP library, however, the protocols and data structures used in the implementation impose restrictions on the input programs.

JCSP does not implement multi-synchronisation, that is, synchronisation involving three or more processes. As *Circus* includes this feature, the translation strategy makes use of a protocol that implements multi-synchronisation. For *JCircus* we designed the class `GeneralChannel` that represent channels; it encapsulates the protocol, in the case of channels involved in multi-synchronisation. This class can be regarded as an extension of JCSP, as its use is not restricted to Java implementations of *Circus* programs.

The main purpose of *JCircus* is to provide animation for *Circus* programs; for that, it also provides a simple graphical interface for execution of the generated programs. We do not have efficiency as a primary concern, but rather, correctness: we have formally verified part of the translation strategy, namely, the multi-synchronisation protocol.

In the next section we give a brief introduction to *Circus* and in Section 3 we present the JCSP library. In Section 4, we present *JCircus* and the translation strategy with a simple example, and we also present the class `GeneralChannel`. In Section 5, we discuss our improvements to the strategy, some errors that were found in the original strategy, the limitations of *JCircus*, and our approach to prove its correctness. In Section 6, we draw conclusions and discuss some directions of future work.

2 *Circus*

In *Circus*, just as in CSP, a system is regarded as a process. However, in *Circus* a process may contain an internal state, which is described using the schema constructs of Z. The state of a process is encapsulated; channels are the only means of communication between a process and its environment.

Like a Z specification, a *Circus* program is formed by a sequence of paragraphs. We use a small example of a program that calculates the greatest common divisor (GCD) between two natural numbers (Figure 1) to explain some of the main constructs of *Circus*.

Our example begins with the declaration of two channels that communicate natural numbers. The channel *in* receives two numbers, in sequence, and the channel *out* outputs their GCD.

A process declaration gives its name and a process definition. The most basic form of process definition specifies the state of the process, a sequence of process paragraphs, and a nameless main action which describes the behaviour of the process. All these are delimited by the keywords **begin** and **end**.

$$
\begin{array}{l}
\textbf{channel}\ in, out : \mathbb{N} \\
\\
\textbf{process}\ GCD\_Euclidean \mathrel{\widehat=} \textbf{begin} \\
\quad \textbf{state}\ GCDState \mathrel{\widehat=} [a, b : \mathbb{N}] \\
\quad InitState \mathrel{\widehat=} x, y : \mathbb{N} \bullet a, b := x, y \\
\quad UpdateState \mathrel{\widehat=} a, b := b, a \bmod b \\
\quad GCD \mathrel{\widehat=} \mu\, X \bullet \textbf{if}\ b = 0 \rightarrow\ out!a \rightarrow Skip \\
\qquad\qquad\qquad\qquad [\!]\ b \neq 0 \rightarrow\ UpdateState;\ X \\
\qquad\qquad\qquad\qquad \textbf{fi} \\
\quad \bullet\ in?x \rightarrow in?y \rightarrow InitState(x, y);\ GCD \\
\textbf{end} \\
\\
\textbf{channel}\ gcd, sum \\
\textbf{channel}\ read, write : \mathbb{N} \\
\\
\textbf{process}\ SumOrGCD \mathrel{\widehat=}\ (GCD\_Euclidean\ [\![\ \{\!|\ in, out\ |\!\}\ ]\!]\ GCDClient) \setminus \{\!|\ in, out\ |\!\} \\
\\
\textbf{process}\ GCDClient \mathrel{\widehat=}\ \textbf{begin} \\
\quad ReadValue \mathrel{\widehat=} read?x \rightarrow read?y \rightarrow ChooseOper(x, y) \\
\quad ChooseOper \mathrel{\widehat=} x, y : \mathbb{N} \bullet \\
\qquad gcd \rightarrow in!x \rightarrow in!y \rightarrow out?r \rightarrow write!r \rightarrow Skip \\
\qquad \Box \\
\qquad sum \rightarrow write!(x + y) \rightarrow Skip \\
\quad \bullet\ \mu\, X \bullet ReadValue;\ X \\
\textbf{end}
\end{array}
$$

Fig. 1. Concrete *Circus* program for calculation of the GCD

In our example, we declare a process *GCD_Euclidean* which has its state described by the schema *GCDState*; it contains two components, a and b, which are initialised with the numbers for which we want to calculate the GCD. The following definitions in the basic process describe actions. The initialisation operation is *InitState*, which defines a parametrised action that takes x and y as input, and assign them to a and b. The action *UpdateState* updates the values of the state components in each iteration of the calculation of the GCD. The recursive action *GCD* implements the Euclidean algorithm for calculation of the GCD. When $b \neq 0$, it recurses; if $b = 0$, then the GCD is output. The basic action *Skip* terminates without communicating values or changing the state.

The main action describes the behaviour of the process. It receives two inputs through channel *in*, initialises the state with these values, and then calls *GCD*.

A process definition like that of *GCD_Euclidean* uses Z and CSP constructs to define the state and the behaviour of the process. It is also possible to define processes in terms of others previously defined, using the CSP operators for sequence, external choice, internal choice and parallelism, among others. The process *SumOrGCD* is a parallel composition of the processes *GCD_Euclidean* and *GCDClient*. They communicate via *in* and *out*, which are hidden; this means that the environment cannot see communications that occur through them.

The process *GCDClient* is recursive: in each iteration, it reads values x and y from a channel *read*, and passes them to the parametrised action *ChooseOper*, which offers a choice between the sum and the greatest common divisor operations. The external choice operator is as in CSP: it offers the environment a choice between two or more actions. If the GCD operation is chosen, it delegates to process *GCD_Euclidean* the calculation of the greatest common divisor; communication occurs through channels *in* and *out*. Otherwise, it outputs on *write* the summation of the two values.

In Section 4, we discuss the translation of this example to Java using *JCircus*. More details on *Circus* can be found in [8].

3 JCSP

The translation strategy makes use of the JCSP library to implement many of the CSP constructs used by *Circus*. The library provides a simplified way to program concurrency in Java without having to deal directly with the Java primitives.

In JCSP, a process is a class that implements the interface `CSProcess`, which defines only the method `public void run()`. The implementation must encode in this method the behaviour of the process.

JCSP also defines interfaces for channels: `ChannelInput` is the interface for input channels and defines the method `read`; `ChannelOutput` is the interface for output channels and defines the method `write`; `Channel` extends both `ChannelInput` and `ChannelOutput` and is used for channels which are not specified as input or output channels. The implementations for channels are the classes `One2OneChannel`, `One2AnyChannel`, `Any2OneChannel` and `Any2AnyChannel`. The appropriate implementation to be used when creating a channel depends on whether there are one or more possible readers and writers for the channel.

Synchronisation in JCSP is not in exact correspondence with the original concept in CSP. Despite being possible to have more than one process that read or write on a channel, only one pair of processes can synchronise at each time; this model is similar to that of occam [9]. Thus, multi-synchronisation, that is, three or more processes synchronising on a single communication, which is allowed in CSP, is not directly supported by JCSP. To solve this problem, the translation strategy implements a protocol for multi-synchronisation.

The class `Alternative` implements the external choice operator. Its constructor takes an array of channels that may be selected. The implementation of the alternation requires that only input channels that have at most one reader participate. The method `select()` waits for one or more channels to become ready to communicate, makes an arbitrary choice between them, and returns the index of the selected channel.

Parallelism is implemented by class `Parallel`, which implements `CSProcess`. The constructor takes an array of `CSProcess`es, which are the processes that compose the parallelism. The method `run` executes all processes in parallel and terminates when all processes terminate. Differently from CSP, it is not possi-

ble to choose the channels on which the processes synchronise; in JCSP, they synchronise on all channels that they have in common.

The CSP constructors *Skip* and *Stop* are implemented by the classes `Skip` and `Stop`, respectively. JCSP includes also implementations for other features that are not available in CSP, such as barrier synchronisation, timers and process managers, among others, and extensions for the `java.awt` library that provide channel interfaces for graphical components. For details, see [6].

4 *JCircus*

The translator from *Circus* to Java is an implementation of the translation strategy that was originally described in [5]. The strategy defines rules for translation of each construct of *Circus*. Translation is carried out by the recursive application of the translation rules, following the syntactic structure of the program. We proposed some adaptations to the original translation strategy, which we discuss later on in this section and in Section 5. The complete reference to the rules implemented in *JCircus* can be found in [10].

JCircus translates a concrete *Circus* program (written in LaTeX markup) into a Java program that implements the specification. It requires from the user: the path of the input file, the name of the project (which will be the name of the Java package for the program), and the output path. Before translation, the tool performs parsing and typechecking, and verifies if the specification meets the requirements for translation.

For each process definition in the input file, the tool asks if the user wants to create a main class for it. For a process X, this class is called `Main_X`, and it is the starting point for the execution of the process. It implements a parallel composition of the process and a graphical interface that simulates its environment. A batch file `Run_X.bat` is also created; it contains commands to compile the project and run the class `Main_X` using JDK [11].

4.1 The translation strategy

The translation consists of two phases. The first phase collects information about types and channels: the free types defined in the program, the channels used by each process, how they are used (for input or output), and whether they are hidden or not. The second phase uses this information to generate the Java code; it is basically an application of the translation rules.

Figure 2 shows as an example the rule for translation of a process declaration. The function $[\![\_]\!]^{ProcDecl}$ is applied to a process declaration ($\mathsf{ProcDecl}$) and takes as parameter the name of the project (N). Each process declaration is translated to a Java class that implements the interface `CSProcess`, and has the same name as the process. The body of the class is translated with the rule for process definition ($\mathsf{ProcDef}$), which we omit here. This rule introduces the attribute declarations, the constructor, and the implementation of the method `run`.

Rule A.1 *Process declaration*

$$[\![\_]\!]^{ProcDecl} : \mathsf{ProcDecl} \pfun \mathsf{N} \pfun \mathsf{JCode}$$

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

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

Fig. 2. Translation rule

Figure 3 shows the class `GCD_Euclidean` (without package and import declarations), which results from the translation of the process *GCD_Euclidean*. Its private attributes are the channels that this class uses: *in* and *out*. As they are not hidden in the declaration of the process, they are taken as input by the constructor of the class. The channels are implemented by the class `GeneralChannel`. The use of this class was one of the modifications to the original strategy, which used the `Any2OneChannel` class provided by JCSP instead.

The translation of the process definition gives the implementation of the method `run` (Figure 3, lines 6-47) of the class. In our example, it is a definition of a basic process, which is translated to a call to the method `run` of an anonymous instantiation of `CSProcess` (lines 7-46).

The anonymous instantiation of `CSProcess` declares the state components as private attributes. Since an action cannot be referenced outside the process where it is defined, action definitions are translated as private methods.

The parametrised action definition *InitState* yields a parametrised method with the same name. The *Circus* multiple assignment is translated to a sequence of Java assignments. The implementation of the multiple assignment in action *UpdateState* needs auxiliary variables because variable a is being updated and used within the same assignment.

The definition of the action *GCD* uses the recursive operator μ; the translation defines an inner class `I_0`. The translation of the recursive action yields the declaration (lines 18-34), initialisation (line 35), and execution (line 36) of the method `run` of this class. It implements `CSProcess`, and its method `run` contains the implementation of the body of the recursion. In the places where a recursive call is made, there is a new instantiation and execution of `I_0` (lines 28-29).

The main action of the basic process is translated as the body of the method `run` for the anonymous class that implements it (lines 39-44). In our example, we have two inputs on channel *in*, a call to *InitState*, and a call to *GCD*.

Figure 4 shows the translation of the process *SumOrGCD*. It is a parallel composition of two other processes; so, its attributes are the channels used by

```
public class GCD_Euclidean implements CSProcess {                    (1)
  private GeneralChannel in, out;                                     (2)
  public GCD_Euclidean(GeneralChannel in, GeneralChannel out) {       (3)
    this.in = in;    this.out = out;                                  (4)
  }                                                                   (5)
  public void run () {                                                (6)
    (new CSProcess() {                                                (7)
      private CircusNumber a, b;                                      (8)
      private void InitState(CircusNumber x, CircusNumber y) {        (9)
        a = x;    b = y;                                            (10)
      }                                                             (11)
      private void UpdateState() {                                  (12)
        CircusNumber aux_a = b;                                     (13)
        CircusNumber aux_b = a.mod(b);                              (14)
        a = aux_a;    b = aux_b;                                    (15)
      }                                                             (16)
      private void GCD() {                                          (17)
        class I_0 implements CSProcess {                            (18)
          public I_0() {}                                           (19)
          public void run() {                                       (20)
            if ((b.getValue() ==                                    (21)
                (new CircusNumber(0)).getValue())) {                (22)
              out.write(a);                                         (23)
              (new Skip()).run();                                   (24)
            } else if (b.getValue() !=                              (25)
                (new CircusNumber(0)).getValue()) {                 (26)
              UpdateState();                                        (27)
              I_0 i_0_0 = new I_0();                                (28)
              i_0_0.run();                                          (29)
            } else {                                                (30)
              while(true){}                                         (31)
            };                                                      (32)
          }                                                         (33)
        }                                                           (34)
        I_0 i_0_0 = new I_0();                                      (35)
        i_0_0.run();                                                (36)
      }                                                             (37)
      public void run() {                                           (38)
        {  CircusNumber x = (CircusNumber) in.read();               (39)
           {  CircusNumber y = (CircusNumber) in.read();            (40)
              InitState(x, y);                                      (41)
              GCD();                                                (42)
           }                                                        (43)
         }                                                          (44)
      }                                                             (45)
    }).run();                                                       (46)
  }                                                                 (47)
}                                                                   (48)
```

Fig. 3. Translation of process *GCD_Euclidean*

```
public class SumOrGCD implements CSProcess {

  private GeneralChannel  gcd, read, sum, write, in, out;
  public SumOrGCD(GeneralChannel gcd, GeneralChannel read,
                        GeneralChannel sum, GeneralChannel write) {
    this.gcd = gcd;
    this.read = read;
    this.sum = sum;
    this.write = write;

    ChannelInfo inf_in = new ChannelInfo();
    inf_in.put("GCDClient", new Integer(0));
    inf_in.put("GCD_Euclidean", new Integer(1));
    this.in = new GeneralChannel(new Any2OneChannel(),inf_in,"SumOrGCD");

    ChannelInfo ch_out = new ChannelInfo();
    inf_out.put("GCDClient", new Integer(1));
    inf_out.put("GCD_Euclidean", new Integer(0));
    this.out = new GeneralChannel(new Any2OneChannel(),inf_out,"SumOrGCD");
  }
  public void run(){
    new Parallel(new CSProcess[] {
      new GCDClient(new GeneralChannel(gcd, "GCDClient"),
                        new GeneralChannel(in, "GCDClient"),
                        new GeneralChannel(out, "GCDClient"),
                        new GeneralChannel(read, "GCDClient"),
                        new GeneralChannel(sum, "GCDClient"),
                        new GeneralChannel(write, "GCDClient")),

      new GCD_Euclidean(new GeneralChannel(in,"GCD_Euclidean"),
                        new GeneralChannel(out,"GCD_Euclidean"))
    }).run();
  }
}
```

Fig. 4. Translation of process *SumOrGCD*

each of them. However, since *in* and *out* are hidden in this process, they are not taken by the constructor, instead, they are created there.

The constructor of the class `GeneralChannel` takes an `Any2OneChannel` and an object of type `ChannelInfo`. This class is a mapping that associates a process name with an integer, that indicates if the instance of the channel is used as an input (1) or an output (0) channel. In our example, channel *in* is used for input by the process *GCD_Euclidean* and for output by the process *GCDClient*; the channel *out* is used for output by *GCD_Euclidean* and for input by *GCDClient*. The constructor also takes the name of the process that is using the instance of the channel; in our case, the process *SumOrGCD*.

The body of the method `run` contains the translation of the parallelism, which uses the class `Parallel` of JCSP. As said before, the constructor of `Parallel` takes an array of `CSProcesses`; in this case, instances of `GCD_Euclidean` and `GCDClient`. Their constructors take the channels used by each process. We construct new instances of the channels based on the ones that we already have, changing only one attribute: the name of the process.

4.2 Running the program generated by *JCircus*

Besides the classes specified by the translation strategy, *JCircus* also generates a simple graphical interface to simulate the execution of a process. For a process X, this class is called `Gui_X`; it represents the environment that interacts with the process. It is a Java Swing frame, with buttons and fields to represent the interface of the process to its environment, namely, the channels that the process uses and are not hidden. The state, hidden channels, and internal operations are not visible by the environment. The graphical interface is also an implementation of a `CSProcess`. It runs in parallel with the class that represents the process.

Figure 5 shows the graphical interface for the process *GCD_Euclidean*. This process uses only two channels: the input channel *in* and the output channel *out*, and both communicate natural numbers. In the text fields next to the buttons, we can type values for the input channels or visualise the values communicated through the output channels.

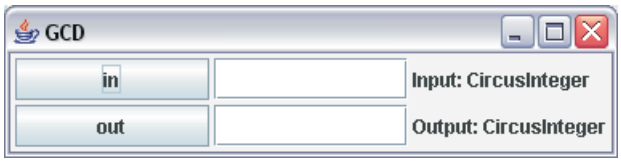

Fig. 5. GUI for process *GCD*

When we run the class `Main_GCD_Euclidean`, the screen presented in Figure 5 is shown. The program waits for a synchronisation on channel *in*, as this is the first communication in which the process *GCD_Euclidean* can engage. As this is an input channel, we must type in the first text field the input value, which is the first of the pair of numbers for which we want to calculate the *GCD*. After entering the value, we press the button `in`; this represents the synchronisation on channel *in*. The generated program does not perform parsing or type checking. It expects that the values entered by the user are well-formed and well-typed.

Once the first number has been entered, the program waits for the second communication through channel *in*. After that, the program calculates the GCD and waits for synchronisation on the channel *out*. When we press the button `out`, the GCD appears in the text field just next to it.

4.3 The class `GeneralChannel`

The original translation strategy used the class `Any2OneChannel` from the JCSP library to implement a simple synchronised channel, that is, a channel on which at most two processes synchronise. This class implements the interface `Channel` which defines methods `read` and `write`. Every synchronisation is point-to-point; it occurs by monitor synchronisation when one reference calls `read` and the other calls `write`. Multi-synchronisation is not directly supported by JCSP.

To allow the implementation of processes that use multi-synchronisation, the original translation strategy makes use of a protocol: for each channel involved in a multi-synchronisation, there is a controller process running in parallel with the system, and this process manages the requests for multi-synchronisation on the channel it controls. The communication between the controller and each process is done via simple synchronisations. Basically, each channel c is replaced by an array of `Any2OneChannel`s `from_c`, which the controller uses to communicate with their clients, and an `Any2OneChannel` channel `to_c`, which is shared by the clients to send messages to the controller.

We have designed a class `GeneralChannel` to provide an abstraction for channels irrespectively of their use in multi-synchronisations. This class encapsulates the data necessary for the implementation of multi-synchronisation; it contains an array of `Any2OneChannel`s `from` and one `Any2OneChannel to`. If the instance of the channel is not involved in a multi-synchronisation, the channel `to` is used, and the array `from` is ignored.

An object of class `GeneralChannel` contains not only the data necessary to implement communication, but it also defines how the channel is used by a process, that is, whether the process writes to or reads from the channel. This information is registered in the attributes `ChannelInfo channelInfo` and `String processName`. The class `ChannelInfo` is a mapping from `String`s to integers. The `String`s are the names of all processes that synchronise on the channel, and, for each of them, the associated integer determines whether the channel is used for writing or reading. The attribute `String processName` records the name of the process that uses the instance of the channel. By looking up `processName` in `channelInfo`, the constructor sets up the attribute `int rw`, which holds 0 if this instance is used for writing, and 1 if it is used for reading.

The class `GeneralChannel` has three constructors: one for channels used in simple communications, one for channels involved in multi-synchronisations, and one that constructs a channel from another `GeneralChannel`, changing only the process name. The last one is used when calling the constructor within a compound process' class; for example, the instantiation of `GCDClient` and `GCD_Euclidean` in Figure 4. When using this constructor, the status `rw` is properly set for the new `GeneralChannel` instance, according to the new `processName`. The signatures of the constructors are presented in Figure 6.

Like the class `Any2OneChannel`, the class `GeneralChannel` defines methods `read` and `write`. In the case of a channel that is not involved in a multi-synchronisation, the implementation just calls the method `read` or `write` of

```
/* Constructor for multi-synchronisation */
public GeneralChannel (Any2OneChannel to, Any2OneChannel[] from,
        ChannelInfo channelInfo, String procName) { ... }

/* Constructor for single-synchronisation */
public GeneralChannel (Any2OneChannel[] from,
        ChannelInfo channelInfo, String procName) { ... }

/* Constructs a new instance of a channel, changing the processName */
public GeneralChannel (GeneralChannel gc, String procName) { ... }
```

Fig. 6. Constructors of `GeneralChannel`

the channel `to`; in the case of a channel involved in multi-synchronisation, these methods contain an implementation of the protocol for the appropriate case.

Besides the methods `read` and `write`, the class `GeneralChannel` also defines the method `synchronise`. It is used in the translation of channels that do not communicate values, as channels *gcd* and *sum* in our example (these channels do not contain input (?) or output (!) fields). Since, in JCSP, the synchronisations are point-to-point, it is necessary to always define a reader and a writer. In our implementation we use the method `synchronise`, so that we do not have to determine if a channel should be read or written, in a particular process.

To determine if a channel is used as a reader or a writer in one process requires inspecting the uses of the process. In the example below we have three processes A, B, and C that execute an event c. These processes are combined in parallel, two at a time in the processes $ParAB$, $ParBC$ and $ParAC$.

$$\textbf{process}\ A \mathrel{\widehat{=}} \textbf{begin}\ \bullet\ c \rightarrow \mathit{Skip}\ \textbf{end}$$
$$\textbf{process}\ B \mathrel{\widehat{=}} \textbf{begin}\ \bullet\ c \rightarrow \mathit{Skip}\ \textbf{end}$$
$$\textbf{process}\ C \mathrel{\widehat{=}} \textbf{begin}\ \bullet\ c \rightarrow \mathit{Skip}\ \textbf{end}$$

$$\textbf{process}\ ParAB \mathrel{\widehat{=}} A \ [\![\{\!|\ c\ |\!\}]\!]\ B$$
$$\textbf{process}\ ParAC \mathrel{\widehat{=}} A \ [\![\{\!|\ c\ |\!\}]\!]\ C$$
$$\textbf{process}\ ParBC \mathrel{\widehat{=}} B \ [\![\{\!|\ c\ |\!\}]\!]\ C$$

If we determine, for instance, that channel c will be a reader in process A and a writer in process B (for the parallelism $ParAB$), then we would not be able to determine the role of channel c in process C; $ParAC$ would require that it was a writer, and $ParBC$ would require it to be a reader.

In our solution, the communications on channel `c` are translated using the method `synchronise`, whose implementation is presented in Figure 7. The attribute `rw` is set in the constructor of the `GeneralChannel`, as explained before, according to the mapping in the `ChannelInfo` objects, which are initialised in the classes `Main_ParAB`, `Main_ParAC` and `Main_ParBC`, with specific mappings for each parallelism.

```
public Object synchronise(Object x) {
    Object r = null;
    if (this.rw == GeneralChannel.READ)
        r = this.read();
    else
        this.write(x);
    return r;
}
```

Fig. 7. Implementation of method `synchronise`

5 Discussions

Our main contribution to the original translation strategy was the class introduced in the last section, `GeneralChannel`. In this section, we discuss other improvements to the original translation strategy, including the correction of an error related to parallelism of actions. Furthermore, we discuss the limitations of *JCircus* and our approach for validating the multi-synchronisation protocol.

5.1 Translation of *Circus* types

The treatment of types in *JCircus* is different from that in the original proposal, in which free types and special forms of abbreviation are translated to classes that represent types. The forms of abbreviation considered were those that defined sets in terms of at most one other set, by extending or restricting its elements; that is, they could have the form $TName_{exp} == TName \cup \{V_1, \ldots, V_m\}$ or $TName_{exp} == TName \setminus \{V_1, \ldots, V_n\}$. For instance, the following example is taken from a case study presented in [8].

$$Mode ::= automatic \mid manual \mid disabled$$
$$SwitchMode == Mode \setminus \{disabled\}$$

In the original translation strategy, these definitions yield two classes: `Mode`, which defines constants `final int automatic = 0`, `final int manual = 1`, and `final int disabled = 2`; and `SwitchMode`, which extends `Mode` and defines a constant `int final MAX = 1`, that restricts the values that it can assume.

We found this approach inappropriate because the notion of type here does correspond to the *Circus* type system, which follows that of Z: *SwitchMode* does not introduce a new type. It actually defines a set; a variable declared as, for instance, **var** $x : SwitchMode$ actually has type *Mode*. The treatment of types in the original translation strategy could result in a situation in which a correctly typed *Circus* program would result in a Java program that does not compile.

In the implementation of *JCircus*, we opted for following the *Circus* type system to have a 1-1 mapping between *Circus* types and Java classes that represent

types. At the moment, we implement only free types and the basic type $\mathbb{A}$; we do not treat compound types yet, which is left as future work. Each free type definition generates a class that represents that type, and abbreviations are not considered. The basic type $\mathbb{A}$, defined in the Z Standard to represent number types, is implemented using class `CircusNumber`, but we have another restriction here, since at the moment this class only implements a subset of $\mathbb{A}$: the set of integer numbers. Although in a *Circus* specification a number variable can hold a value from an infinite set, in a programming language, like Java, we have finite memory, which restricts the actual ranges that can be represented.

5.2 Parallelism of actions

We found a mistake regarding the translation of action parallelism. Action parallelism is different from process parallelism because the former requires the definition of the set of variables that each parallel action can modify; we call this set a partition. For process parallelism, there is no such concern, since each process can access only its own data. The semantics of action parallelism defines that each parallel action deals with copies of the local variables, and at the end of the parallelism, the original variables are updated with the values of their respective copies from the actions where they appear in the partition. In this way, concerns about shared access to the state by the parallel actions are avoided.

The original translation strategy did not reflect this semantics when one of the parallel actions contained an action call. In this situation, the action call, which was translated as a method call, would update the original values, instead of the copy. Our solution was to change the translation of an action call that occurs within a parallelism; it is translated using an inner class (that implements `CSProcess`) that declares as attributes copies of the local variables. The translation consists of instantiation and execution of this class, and update of the values of the original variables at the end.

5.3 Limitations

The implementation of *JCircus* also helped us to identify some requirements that were not explicitly stated in the original translation strategy. We describe three of them here. First, synchronisation on channels must always involve the same number of processes, and the same processes. So, this is not allowed.

$$\textbf{process}\ P_1 \mathrel{\widehat=} \textbf{begin}\ \bullet\ c \rightarrow \mathit{Skip}\ \textbf{end}$$
$$\textbf{process}\ P_2 \mathrel{\widehat=} \textbf{begin}\ \bullet\ c \rightarrow \mathit{Skip}\ \textbf{end}$$
$$\textbf{process}\ P_3 \mathrel{\widehat=} \textbf{begin}\ \bullet\ c \rightarrow \mathit{Skip}\ \textbf{end}$$

$$\textbf{process}\ P \mathrel{\widehat=} (((P_1 \llbracket \{\!| c |\!\} \rrbracket P_2) \llbracket \{\!| c |\!\} \rrbracket P_3);\ (P_1 \llbracket \{\!| c |\!\} \rrbracket P_2)) \setminus \{\!| c |\!\}$$

Process P is a sequential composition; the first process is a parallelism of three processes P_1, P_2 and P_3, synchronising on c; the second is a parallelism of P_1 and P_2, synchronising on c. So, in the first parallelism, channel c is involved in a multi-synchronisation, whereas in the second one, it is not. This cannot occur; the

channel `c` is instantiated in the constructor of class `P`, and the settings regarding multi-synchronisation must hold for the whole implementation of process `P`.

The second limitation we discuss here also involves parallelism. Because the implementation of class `GeneralChannel` uses the name of the processes involved to determine how each process uses the channel, there cannot be repeated copies of a processes in a parallelism, or parallelism of anonymous processes, like in the following examples.

$$\textbf{process}\ P \mathrel{\widehat=} A \,[\![\, \{\!] \ \ldots\ [\!\}\,]\!]\, A$$
$$\textbf{process}\ Q \mathrel{\widehat=} \textbf{begin}\ \ldots\ \textbf{end}\, [\![\, \{\!]\ \ \ldots\ \ [\!\}\,]\!]\, \textbf{begin}\ \ldots\ \textbf{end}$$

However, this is not a serious restriction, which can be solved by redefining the processes with new names. This rewriting could be automatically done by *JCircus*, and this is one of the improvements that are planned for future versions.

The third limitation is that the situation in which a channel is used for reading and for writing by the same process is not allowed. The reason is the design of the class `GeneralChannel`, already discussed; it requires that the role of a channel is uniquely determined in each process where it is used.

These and other limitations are recognised and documented as restrictions on the input specification. They are also checked by *JCircus*; it gives an error message in the case that the input program violates one of these restrictions.

5.4 Verification of the multi-synchronisation protocol

In order to have a useful tool, we are concerned not only with the correct implementation of the translation rules, but also with a guarantee that the rules themselves are correct. A complete proof of soundness for the translation strategy requires a formal semantics for Java, and a mapping from the *Circus* semantics. With that, we could prove that the semantics of every *Circus* program is in correspondence with the semantics of the Java program obtained with the translation. This, however, is by no means a simple task. We have proposed a smaller step to bridge the gap between *Circus* and Java: to model the JCSP constructs and the Java programs in *Circus* itself, and use the *Circus* refinement calculus to prove that the translation rules are refinement laws. We used this approach to tackle the verification of the algorithm for multi-synchronisation.

We were inspired by the work on [12] which considers a simple form of multi-synchronisation. It is verified to be equivalent to another model that uses only simple synchronisations. We have used a similar approach to verify a more complex type of multi-synchronisation, where the channel takes part in an external choice. We proposed a *Circus* model for this kind of multi-synchronisation; then we proposed a *Circus* model for the multi-synchronisation protocol, and proved, using the refinement calculus of *Circus* that the multi-synchronisation is refined by our model of the implementation. The model is very close to the implementation and improves our confidence in it.

The approach taken for carrying out the refinement consists in refining the specification to an action system; transforming the implementation model into

another action system; and then proving that the action systems are equivalent. An action system is a type of recursive process whose execution is controlled by a local variable which determine which events that are enabled in each iteration. We have used equality or refinement rules in each step of the transformation.

6 Conclusions

We have described *JCircus*, a tool that implements a translation strategy from *Circus* to Java. *JCircus* was developed in Java itself, and the translator module amounts to about 10000 lines of code. We have followed a structured approach for design and testing, and the project is documented in UML.

JCircus was implemented using the CZT framework [13], an open-source Java framework for the ISO Z Standard and extensions. It has been recently extended to support *Circus*. The framework provides, among other things, a Java library for abstract syntax trees, basic tools like parsers, type checkers and printers, and an interchange format, based on XML, for representing specifications. The use of the CZT framework allows future integration with other tools for *Circus* that use the same framework. Currently, we have a parser and a type checker [14] for *Circus*, that *JCircus* already uses, and a prototype model checker for *Circus* [15]. A long-term goal of the *Circus* group is to provide an integrated environment that supports development using *Circus*. *JCircus* is available at `http://www.cs.york.ac.uk/circus/jcsp/freitas-msc/`, where we can also find examples of translations.

Some previous work on translation tools are a translator from a subset of CSP into Handel-C [16], and a translator of CSP to Java and C code [17]. The latter also uses libraries that implement CSP operators in a programming language. Their strategy is similar to ours, however, we have the translation rules formalised and we cover a broader range of CSP operators. As far as we know, there is no other translator for *Circus*.

Another distinguishing feature of our work is the generation of the graphical interface. It is an additional functionality provided by *JCircus* and is not formalised by the translation rules. It makes the execution of the program generated immediately available, and is appropriate for the rapid prototyping of *Circus* programs. The classes that capture the behaviour of each of the processes, however, can be used in other contexts, where, for example, an interface that is more specific to the application is implemented.

JCircus can translate some interesting examples, but the work is far from complete. There are still some features that need to be implemented to make the tool more useful. Some *Circus* constructs are not supported by our tool because we do not have a robust parser yet. However, the translation rules are all implemented. The CZT project is currently working on a new parser for *Circus*, which extends the existing Z parser. When this parser is done, the translation of the constructs which are not currently supported will become available.

One important extension is the provision of implementation of compound *Circus* types, and number types other than integers in the `CircusNumber` class.

Implementation of more types will make the tool available for translation of a larger range of programs. Another piece of future work is the investigation of new implementations for the external choice and parallelism that avoid the limitations of classes `Alternative` and `Parallel` of JCSP. The limitations described in Section 5.3 also make an interesting topic for future research.

Acknowledgements

The authors thank Marcel Oliveira for extensive discussion about the translation strategy; Leo Freitas for discussion about the design of *JCircus*; and Manuela Xavier, who implemented the typechecker which is part of *JCircus*. This work has been partially funded by the Royal Society and QinetiQ.

References

1. J. C. P. Woodcock and A. L. C. Cavalcanti. A Concurrent Language for Refinement. In *5th Irish Workshop on Formal Methods*, 2001.
2. J. C. P. Woodcock and J. Davies. *Using Z – Specification, Refinement, and Proof*. Prentice-Hall, 1996.
3. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
4. C. A. R. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.
5. M. V. M. Oliveira. *A Refinement Calculus for Circus*. PhD thesis, Department of Computer Science, The University of York, 2005.
6. P. H. Welch. Process Oriented Design for Java: Concurrency for All. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2000)*, 2000.
7. Z Standards Panel. Formal Specificationn – Z Notation – Syntax, Type and Semantics — Consensus Working Draft 2.6, 2000. At http://www.cs.york.ac.uk/ ian/zstan/.
8. M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. Refining Industrial Scale Systems in *Circus*. In *Communicating Process Architectures*, 2004.
9. The occam archive. At http://vl.fmnet.info/occam/.
10. A. Freitas. From *Circus* to Java: Implementation and Verification of a Translation Strategy. Master's thesis, Department of Computer Science, The University of York, 2005.
11. Java Development Kit. http://java.sun.com/javase/.
12. J. C. P. Woodcock. Using *Circus* for Safety-Critical Applications. In *VI Brazilian Workshop on Formal Methods*, 2003.
13. P. Malik and M. Utting. CZT: A Framework for Z Tools. In *5th International Conference of Z and B Users (ZB 2005)*, 2005.
14. M. Xavier. Definição Formal e Implementação do Sistema de Tipos para a Linguagem *Circus*. Master's thesis, Centro de Informática, Universidade Federal de Pernambuco, Brazil, 2006. To be submitted.
15. L. Freitas. *Model checking Circus*. PhD thesis, Department of Computer Science, The University of York, 2005.
16. J. D. Phillips and G. S. Stiles. An Automatic Translation of CSP to Handel-C. In *Communicating Process Architectures*, 2004.
17. V. Raju, L. Rong, and G. S. Stiles. Automatic Conversion of CSP to CTJ, JCSP, and CCSP. In *Communicating Process Architectures*, 2003.