Communicating Process Architectures 2012
P.H. Welch et al. (Eds.)
Open Channel Publishing Ltd., 2012
© *2012 The authors and Open Channel Publishing Ltd. All rights reserved.*

JCircus 2.0: an Extension of an Automatic Translator from *Circus* to Java

S. L. M. BARROCAS and M. V. M. OLIVEIRA [1]

Universidade Federal do Rio Grande do Norte, Brazil

Abstract. The use of formal methods in the development of concurrent systems considerably reduces the complexity of specifying their behaviour and verifying properties that are inherent to them. Development, however, targets the generation of executable programs; hence, translating the final specification into a practical programming language becomes imperative. This translation is usually rather problematic due to the high probability of introducing errors in manual translations: the mapping from some of the original concepts in the formal concurrency model into a corresponding construct in the programming language is non-trivial. In recent years, there is a growing effort in providing automatic translation from formal specifications into programming languages. One of these efforts, *JCircus*, translates specifications written in *Circus* (a combination of Z and CSP) into `Java` programs that use `JCSP`, a library that implements most of the CSP constructs. The subtle differences between `JCSP` and *Circus* implementation of concurrency, however, imposed restrictions to the translation strategy and, consequently, to *JCircus*. In this paper, we extend *JCircus* by providing: (1) a new optimised translation strategy to multi-way synchronisation; (2) the translation of complex communications, and; (3) the translation of CSP sharing parallel and interleaving. A performance analysis of the resulting code is also in the context of this paper and provides important insights into the practical use of our results.

Keywords. *Circus*, automatic translation, Java, JCSP.

1. Introduction

Using formal methods, developers are able to describe the properties of the system under development in a concise, correct, and unambiguous manner. Furthermore, the formal descriptions of the systems are amendable to verification. For this reason, the use of formal methods in the development process increases the reliability of software systems and allows developers to ensure that the resulting system meets a determined set of properties. Development, however, aims at providing executable programs; there exists an intrinsic need to translate formal specifications into programming languages.

Manual translations are very likely to introduce errors due to the inherent complexity of concurrent systems. In the last decade, tool support for formal methods have increasingly added automatic translation into programming languages [5,4,20]. These translators fosters the use of formal methods as they provide a possibility of effortlessly generating executable code from a verified formal specification.

A very well known formal method is Z [28], which is semantically based on the Zermelo/Fraenkel set theory and provides a notation for the specification of state-rich sequential systems. Although possible, the specification on concurrent systems is not within the main

[1]Corresponding Author: *Marcel Vinícius Medeiros Oliveira, Campus Universitário - UFRN*. Tel.: (55) (84) 3215-3814; E-mail: `marcel@dimap.ufrn.br`.

scope of Z. An efficient notation for concurrent systems is Communicating Sequential Processes (CSP) [15] that represents systems as processes that perform events. CSP, however, does not support a concise and elegant way to describe the data aspects of state-rich systems.

The combination of different formalisms allows the reuse of notations in an integrated framework that is able to describe different aspects of the systems. Many formalisms combine data and behavioural aspects of the system. Combinations of Z with CCS [13], Z with CSP [9], Object-Z [2] with CSP [8], and Object-Z with timed CSP [16] are some of these attempts to combine both schools. Further combinations of other state-based formal languages like B with CSP [22] are also available in the literature. *Circus* [27] is a formal language that combines Z and CSP providing an elegant style for the specification of state-rich concurrent systems. It differs from other combination languages in that it also has an associated refinement calculus [18] that allows a calculational correct-by-construction system development.

The *Circus* tool set includes a model-checker [12] and a tool that supports its refinement calculus [19] and its tactic language [7]. All tools are based on the *Circus* parser and type-checker provided by the CZT [1], which were also the basis of *JCircus*, a translator from *Circus* to Java, whose initial version was presented in [10].

JCircus generates Java code that uses `JCSP` [26,24], an API that provides an abstraction interface for most CSP primitives in Java. The concurrency implementation of `JCSP`, however, has some subtle differences from the CSP's concurrency model used in *Circus*. For instance, `JCSP` implements Hoare's parallelism [14] in which processes synchronise in the intersection of their alphabets. On the other hand, *Circus* uses Roscoe's parallel composition in which it is possible to restrict the events on which the processes involved synchronise [21]. For this reason, *JCircus* as presented in [10], imposed restrictions to the parallel composition used in the *Circus* specification given as input. By way of illustration, processes with shared channels could not be composed in interleaving.

This paper describes an extension to *JCircus* that: (1) optimises the translation of multi-synchronisation; (2) includes the translation of complex communications; and (3) removes the restrictions on sharing parallel and interleaving. The optimisation of multi-synchronisation replaces the previously used multi-synchronisation protocol by a new strategy that makes extensive use of the `JCSP`'s `AltingBarrier` [25]. The translation of prefixing is extended with the possibility of translating communications with an arbitrary decoration of fields (the original strategy only translates communications with dot fields, and a possible input or output field as the last field of the communication). Finally, the translation of sharing parallel and interleaving rewrites the specification using CSP's renaming whilst preserving the specification's semantics.

This paper also discusses the integration of *JCircus* with CRefine [19]. This integration provides *Circus* with a framework that can be used to formally develop systems from an abstract specification to a Java executable program. Finally, the paper provides a performance analysis of the resulting code, which indicates the possibility of using *JCircus* in larger developments.

This paper is organized as follows: Section 2 describes *Circus*, the source language of the *JCircus*, which is described in Section 3. The JCSP library is described in Section 4. The main contribution of the paper is presented in Section 5, where we describe the details of our translation of multi-synchronisation, complex communication and interleaving. An analysis on the performance of extensions is presented in Section 6. Finally, Section 7 presents our concluding remarks and discusses future work.

2. *Circus*

Concurrent and Integrated Refinement CalculUS (*Circus*) [27] is a formal language that combines Z, CSP, and guarded commands [6]. With this combination, *Circus* aggregates the power of representing complex data structures in Z with the process algebra of CSP. Furthermore, *Circus* also has a refinement calculus associated to it [3].

A *Circus* program is formed by a sequence of paragraphs. Each paragraph can be a Z paragraph, a channel declaration, a channel set declaration, or a process declaration. *Circus* processes may be explicitly defined or defined in terms of other processes.

For illustration purposes, in Figure 1, we present an example of a *Circus* specification of a casino that specifies a Roulette, with two players (only 50-50 bets), a single table, and its croupier. The global constant *VALUE* is a Z paragraph declared at the top of the specification. All channels used in the specification are declared in the channel declaration paragraphs. The specification has six processes: *Roulette*, *Player*, and *Croupier* are the basic processes. The remaining processes *Table*, *Players*, and *Casino* are defined as compositions of the basic processes.

An explicitly defined process is delimited by the keywords **begin** and **end**. It may have a state (represented by a Z schema), and zero or more *Circus* actions (state operations or CSP-like actions). An explicit defined process, however, must have a declared main action that describes its main behaviour.

By way of illustration, process *Player* represents a player at the casino. Each player is given an identification *id* and has a *bankRoll* that represents the player's initial amount of money. The *Player* state is defined in the Z state schema *PlayerSt*: it is composed by the player's current bet *b* and his amount of *cash*. This state may be changed using the operations *PlayerInit*, *PayBet*, or *WinBet*. The CSP-like action *BRBet* describes the *Player*'s behaviour in a single bet. Finally, the *Player* behaviour is defined in its main action: the state is initialised and the *Player* behaves recursively. In each iteration, the *Player* makes a *BRBet* and terminates if his cash finishes or his winnings are above 100; he recurs and keeps betting otherwise.

Circus has three primitive actions: *Skip*, *Stop* and *Chaos*. The action *Skip* terminates successfully and does not change the state. The second action deadlocks and *Chaos* diverges. The prefixing operator is standard, but a guard construction may be associated with it. For instance, given a Z predicate p, if the condition p is *true*, the action $p \;\&\; c?x \rightarrow A$ inputs a value through channel c and assigns it to the variable x, and then behaves like A, which has the variable x in scope. If, however, the condition p is *false*, the same action blocks. Such enabling conditions like p may be associated with any action. Prefixing may also allow one or more values to be transferred from a process to another (or to other processes, in the case of a multi-synchronisation) by using multiple communication fields, each of which has a decoration: an input decoration (?), an output decoration (!), or a dot decoration (.). For example, for a **channel** $c : \mathbb{N} \times \mathbb{N} \times \mathbb{N}$, the communication $c?x.5!4$ receives any natural number and assigns this value on x, synchronises on 5, and outputs 4. Hence, $c.0.5.4$ is a possible synchronisation.

Circus actions may be composed using the CSP operators of sequence, external and internal choice, parallelism, interleaving and hiding. However, the parallelism and interleaving actions have a different declaration. In order to avoid conflict, they require the declaration of two sets that partition the variables in scope. In the parallel composition $A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2$ the actions A_1 and A_2 synchronise on the channels in the set cs. Both actions, A_1 and A_2, have access to the initial values of all variables in scope. However, A_1 and A_2 may modify only the variables in ns_1 and ns_2, respectively. Interleaved actions $A_1 \interleave\!\![ns_1 \mid ns_2]\!\!\interleave A_2$ have the same behaviour regarding the state variables but they do not synchronise on any channels and run independently.

$VALUE ::= RED \mid BLACK$

channel $start, stopBet$
channel $enter, pay : \mathbb{N}$
channel $bet : \mathbb{N} \times VALUE$
channel $result : VALUE$

process $Roulette \mathrel{\widehat=}$
begin

$$\bullet\ \mu X \bullet start \rightarrow \begin{pmatrix} result.RED \rightarrow X \\ \sqcap\ result.BLACK \rightarrow X \end{pmatrix}$$

end
process $Player \mathrel{\widehat=}\ id : \mathbb{N};\ bankRoll : \mathbb{N} \bullet$
begin

state $PlayerSt \mathrel{\widehat=}\ [b : VALUE;\ cash : \mathbb{N}]$

$PlayerInit \mathrel{\widehat=}\ b, cash := RED, bankRoll$

$PayBet \mathrel{\widehat=}\ cash := cash - 1$

$WinBet \mathrel{\widehat=}\ cash := cash + 2$

$BRBet \mathrel{\widehat=}$

$$\begin{array}{l} enter!id \rightarrow \begin{pmatrix} (b := RED \sqcap b := BLACK); \\ PayBet;\ bet.id!b \rightarrow result?x \rightarrow \\ \qquad \begin{pmatrix} \mathbf{if}\ (x = b) \rightarrow pay.id \rightarrow WinBet \\ [\!]\ (x \neq b) \rightarrow Skip \\ \mathbf{fi} \end{pmatrix} \end{pmatrix} \\ \Box\ stopBet \rightarrow Skip \end{array}$$

$$\bullet\ PlayerInit;\ \mu X \bullet BRBet;\ \begin{pmatrix} \mathbf{if}\ (cash = 0 \lor cash > 100) \rightarrow Skip \\ [\!]\ (cash \neq 0 \land cash \leq 100) \rightarrow X \\ \mathbf{fi} \end{pmatrix}$$

end

process $Croupier \mathrel{\widehat=}$
begin

$StartRoulette \mathrel{\widehat=}\ start \rightarrow TakeBet$

$$\begin{array}{l} TakeBet \mathrel{\widehat=}\ enter?id \rightarrow bet.id?b \rightarrow result?x \\ \qquad \begin{pmatrix} \mathbf{if}\ (x = b) \rightarrow (pay.ident \rightarrow Skip) \\ [\!](x \neq b) \rightarrow Skip \\ \mathbf{fi} \end{pmatrix} \\ \quad \sqcap\ stopBet \rightarrow Skip \end{array}$$

$\bullet\ \mu X \bullet StartRoulette;\ X$

end

process $Table \mathrel{\widehat=}\ Roulette \,[\![\, \{\!|\ start, result\ |\!\}\,]\!]\, Croupier$

process $Players \mathrel{\widehat=}\ Player(0, 10) \,|\!|\!|\, Player(1, 20)$

process $Casino \mathrel{\widehat=}\ Players \,[\![\, \{\!|\ enter, bet, result, pay\ |\!\}\,]\!]\, Table$

Figure 1. An Example of a *Circus* Specification.

In process *Player*, the action *PayBet* specifies the payment of the bet, which decrements the *cash* by one. On the other hand, the *WinBet* increments the *cash* by 1. The CSP-like action *BRBet* initially offers an external choice: the *Player* may either output his *id* on channel *enter*, or successfully terminate after the *stopBet* event. After sending his *id*, the *Player* makes an internal choice on the bet. Next, the *Player* stores the chosen bet value *i* in *b*, pays the bet by invoking *PayBet*, puts the bet on channel *bet*, and receives the *result*. If the *result* is the same as the value of the *bet*, the *Player* receives an indication of payment on channel *pay* and increments his *cash*; the bet simply terminates otherwise.

The process *Croupier* is an instance of a stateless *Circus* process and also has a recursive behaviour. The action *StartRoulette start*s the *Roullette*, and then behaves like the *TakeBet* action. In this action, the *Croupier* internally decides if he will take a bet or close the table. If he decides to take a bet, the *Croupier* allows a player to *enter* the table, takes his *bet* and receives the *result* from the *Table*. If the *Player* made the right bet, the *Croupier pay*s the bet; it simply recurs otherwise. The *Roulette* process is yet another instance of a stateless *Circus* process. Its recursive main behaviour is very simple. The *Roulette* receives the *start* command and internally decides the *VALUE* that will provide on channel *result*.

The CSP operators of sequence, external and internal choice, parallelism, interleaving, their corresponding iterated operators, and hiding may also be used to compose processes. For instance, process *Table* is a parallel composition between a *Roulette* and a *Croupier*. Both processes synchronise on the events *start* and *result*. On the other hand, process *Players* is an interleaving of two players; hence, both players run independently. Finally, the *Casino* is a parallel execution of the *Players* and the *Table* synchronising on *enter*, *bet*, *result* and *play*.

In this paper we extend a strategy for implementing *Circus* programs in JCSP. We assume that, before applying the translation strategy presented in this chapter, the specification of the system we want to implement has been already refined, using the *Circus* refinement calculus to meet the translation strategy's requirements discussed in [18]. The concrete nature of this specification allows a natural translation into most of constructs presented in the next Section. Nevertheless, some restrictions on previous versions of the tool have been removed based on the results presented in Section 5.

3. *JCircus*

JCircus [11][1] is a Java application that translates *Circus* specifications into Java code that uses the JCSP API. It has four modules: the parser is provided by the CZT and returns an Abstract Syntax Tree (AST) of the given specification; the type-checker verifies the type correctness of the specification and adds type information into the AST; the pre-processor visits the AST nodes and collects information from the specification (*e.g.* variables, names, and types) in environments and checks if the specification satisfies the requirements of *JCircus*; and the translator visits the AST nodes and returns a package with the generated Java code. The generated code is distributed in a way such that each process of the specification has a corresponding Java file.

JCircus is based on transformation rules of a translation strategy proposed in [18]. For example, there are rules for translating *Circus* actions into Java methods, and rules for translating *Circus* communications on `c` into `c.read()` or `c.write(null)`. The main constructs of *Circus* are translated as follows:

- Processes are translated into Java classes. The process local variables, state components and visible channels are declared as private attributes in the class and are instantiated in the constructor of the class;

[1]Project Webpage: `http://www.dimap.ufrn.br/~marcel/research/jcircus/`

- *Circus* actions are translated into private methods;
- The main action of a *Circus* process is translated into the run method of the process;
- Communication is translated into an invocation to either the read or the write methods (possibly accessing array positions in the case of more than one value being communicated);
- Multi-synchronisation is translated into a code that implements an extension of the protocol from [18];
- External choice is translated using the Alternative class;
- Internal choice between two actions is implemented using a random choice;
- Parallelism is implemented using the Parallel class (the intersection of the processes alphabets must be a subset of the synchronisation channel set);

JCircus has an extremely simple GUI, composed of a single window, in which the user enters the path of the input specification and the name and path of the generated Java project. Compilation errors, type errors, and non-compliance with the tool's requirements are exhibited in a Log area. If no errors are found, the user is asked to choose the system's main process, after which the code is automatically generated.

Despite being already very useful, the previous version of *JCircus* had a number of restrictions, some of which imposed interesting challenges. In the next section, we discuss the challenges involved in removing some of these restrictions.

4. JCSP

Java Communicating Sequential Processes (JCSP) [26,24] is a Java API that provides a CSP style for programming concurrency. JCSP is implemented on the top of the original Java primitives for concurrency (threads), and its main goal is to simplify the implementation of concurrent systems. A process in JCSP is a class that implements the CSProcess interface. The behaviour of the process is implemented in the method run, and the implementation of CSProcess requires the implementation of this method.

JCSP provides a variety of channel types: Any2OneChannel for many writers and one reader; Any2AnyChannel for many writers and many readers; One2AnyChannel for one writer and many readers; and One2OneChannel for one reader and one writer. Each of these channels has two front-ends: one for input and one for output. For many readers, the input front-end is shared and implemented by SharedChannelInput. On the other hand, if the front-end is not shared, it is implemented by AltingChannelInput. Similarly, output front-ends are implemented by SharedChannelOutput and AltingChannelOutput. Sharing, however, does not correspond to multi-synchronisation and the communication on these channels are point-to-point communications. If two processes try to access a shared channel end at the same moment, an exception is thrown by the JCSP kernel.

Communications are made by invoking the read and write methods of a front-end of a JCSP channel. The available channels of JCSP have only one communication field, which can be an input field or an output field. An input front-end of a channel has an input field, and an output front-end of a channel has an output field.

The external choice is implemented using an alternation (Alternative), which offers a set of guards (Guard - *e.g.* front-ends of alting channels) for synchronisation, and returns the index of the chosen guard. The front-ends that are not shared extend Guard, and can also participate on an alternation.

Parallelism is implemented using the class Parallel. This class implements a process that takes an array of CSP processes (CSProcess) and executes them in parallel. The parallelism supported by JCSP is Hoare's parallelism, in which parallel processes synchronise on the intersection of their alphabets.

In `JCSP`, a `Barrier` corresponds to fundamental multiway event synchronisation in CSP. However, although CSP allows processes to offer multiway events as part of an external choice, `JCSP` does not permit this for `Barrier` synchronisation. In [23], Welch *et al* introduced to `JCSP` the `AltingBarrier` class that removes this constraint, allowing multiple barriers to be included in the guards of an Alternative along with skips, timeouts, channel communications and call channel accepts.

An alting barrier (`AltingBarrier`) has a set of front-ends: there must be one process enrolled in each front-end. The synchronisation of a process on an alting barrier is made either by invoking the `sync` method or by offering the barrier in a choice. Any process synchronising on the alting barrier will be blocked until all processes associated with the alting barrier have synchronised. In case of a choice, however, the other alternatives are also available for synchronisation.

By way of illustration, we present parts of the `JCSP` implementation of the process *Player* and *Players*. As every process in `JCSP`, `Player` is a class that implements `CSProcess`:

```
public class Player implements CSProcess {
```

The parameters and state variables are declared as private attributes of `Player`:

```
private Integer b, cash, id, bankRoll;
```

Each of the process' channel front-ends is also declared as private attributes. In our example, *bet* is declared as an array of `AltingChannelOutput` because it communicates two values: the first value is implemented as the array index and the second value is the actual communication value. The channel *stopBet* is implemented as an alting barrier because of the multi-synchronisation in the original specification.

```
AltingChannelOutput enter, pay;
AltingChannelOutput [] bet;
AltingChannelInput result;
AltingBarrier stopBet;
```

All these attributes are initialised in the class constructor with the values received in the constructors argument. For conciseness, we omit its definition here.

The actions of the *Circus* process are implemented as private methods. The translation of the action *WinBet* is presented below: the translation of arithmetic expressions and assignments is straightforward for variables of basic types.

```
private void WinBet () { this.cash = this.cash + 35; }
```

The *BRBet* initially offers an external choice on *enter* and *stopBet*. Its implementation uses an `Alternative` that is initialised with an array of guards containing these two channels. The `select` method returns the index of the channel chosen in the choice.

```
public void BRBet () {
   Alternative alt =
     new Alternative (new Guard [] { this.enter, this.stopBet });
   int select = alt.select();
```

The value of the variable `select` is 0 if the *enter* event is chosen, and 1 if the *stopBet* event is chosen. A switch block that follows the selection controls the execution flow according to the choice made. If the *stopBet* event is chosen, the implementation skips; otherwise, the translation of the iterated internal choice is executed. This translation makes a random choice of a value within the *RANGE* value and uses this value in the execution of the following actions, which write the value of the bet on channel *bet* and receives the *result*.

```
switch (select){
 case 0:{
     int choosen = RandomGenerator.generateNumber(0, 1);
     switch (choosen){
         case 0:{ b = new VALUE(VALUE.RED); } break;
         case 1:{ b = new VALUE(VALUE.BLACK); } break; };
     PayBet();
     bet[this.id.intValue()].write(b);
     Integer x = (Integer) result.read();
     break; }
 case 1:{ (new Skip()).run(); break; } }
```

JCSP does not allow communications of multiple values. For this reason, communication on *bet* is implemented as an array access on the first value and a standard JCSP communication on the second value.

In implementation of the *Players* process we use the Parallel class as follows.

```
(new Parallel (
  new CSProcess [] {
     new Player (0, 10, enter, bet, result, pay, stopBet),
     new Player (1, 20, enter, bet, result, pay, stopBet),
}
)).run();
```

In this example, we run a Parallel object that executes two different Players, given in the array used as argument in the constructor. Although sharing the same channel ends, such implementation does not present a spurious behaviour because their access to the shared channel *result* is controlled by a exclusive communication on *enter*. *JCircus*, however, treats these channels as multi-synchronised channels as we discuss in the next section and in Section 5.

5. Extensions to *JCircus*

This section explains the extensions done to *JCircus*. There were three main extensions: multi-synchronisation (Section 5.1), complex communication (Section 5.2), and sharing parallel composition and interleaving (Section 5.3). The following sub-sections explain these extensions. For conciseness, we omit some details of the examples presented here. A full account of the examples of this paper, input files, generated code, and the analysis data can be found at www.dimap.ufrn.br/~marcel/research/jcircus.

5.1. Multi-synchronisation

The first main change to *JCircus* is an optimisation in the support of multi-synchronisation. This was previously implemented using a centralised protocol [18] with a controller that deals with multi-synchronisation requests from each participating process. The controller and the processes communicate using point-to-point channels.

The multi-synchronisation protocol, however, has two limitations: (1) multi-synchronisation in a channel must always involve the same number of processes, and (2) a multi-synchronisation must not define more than one writer. The former is due to the way the controller deals with the clients using a counter to ensure all participants are willing to synchronise, and the latter is due to the use of process identification to ensure the writer has the first access to the synchronisation.

The use of the multi-synchronisation protocol in [11] was motivated by the lack of a JCSP built-in constructor for multi-synchronisation on external choices. In [25], however, JCSP was extended with AltingBarriers that behave as synchronisation barriers (multi-synchronisation) and can take part in external choices. Each instance of the class AltingBarrier represents one *front-end* of a whole alting barrier.

The strategy of multi-synchronisation with alting barriers was implemented by encapsulating each *front-end* in GeneralChannel, a class used by *JCircus* that encapsulates the multi-synchronisation protocol. Each instance of a multi-synchronised channel, in the extended version of *JCircus*, has an alting barrier *front-end*.

The translation of multi-synchronisation into JCSP was not achieved, however, by directly using alting barriers because they are not capable of communicating values. Our strategy to use these barriers resulted in an approach that allows value communication in a multi-synchronisation. Our approach translates multi-synchronised channels that communicate n values as n-dimensional GeneralChannel arrays, in which each dimension represents one of the communicated values. For example, a given channel **channel** $c : T_1 \times T_2$ is translated to GeneralChannel [][] c;. Here, c is a bi-dimensional array whose dimensions have sizes $\#T_1$ and $\#T_2$, respectively.

In the example presented in Section 2, three processes (the *Roulette*, the *Croupier* and one of the *Player*s) synchronise on channel *result* making this channel a multi-synchronised one. In this synchronisation, the *Roulette* sends the chosen value to the other two parts.

In Figure 2, we present a sketch of the translation of the Roulette's behaviour. The Roulette initially takes part in the multi-synchronisation on start. Afterwards, it randomly chooses to either offer the result RED or BLACK. In both cases, the Roulette gives the result by taking part in the corresponding multi-synchronisation on result. In the case of the Roulette, the multi-synchronisation $c.i$ is implemented as a simple synchronisation at the i-th element of the array c, where i is the randomly chosen value.

```
public void run(){
  start.sync();
  int choosen = RandomGenerator.generateNumber(0, 1);
  switch (choosen){
     case 0:{
         result[ (new VALUE(VALUE.RED)).getValue()].sync();
         (new Skip()).run(); } break;
     case 1:{
         result[ (new VALUE(VALUE.BLACK)).getValue()].sync();
         (new Skip()).run(); } break;
  };
  ... // Recursion
}
```

Figure 2. Translating Multi-synchronisation with Value Communication - Roulette.

The implementation of the communication at the Player and the Croupier offers all elements of the array of barriers result. This is achieved by offering a synchronisation of all elements of the array result using a Alternative construct.

As we discuss in Section 6, by replacing the use of the protocol by alting barriers we considerably optimised the implementation of multi-synchronisation. Nevertheless, value communication could not be directly translated into channel communication because JCSP barriers do not communicate values. For this, reason we needed to translate value communication as array access as explained above. This implementation, however, motivated a further

extension to *JCircus*: the translation of communications with arbitrary number of fields. In the next section we describe the approach used to translate these complex communications.

5.2. Complex communications

In `JCSP`, a communication may only transmit a single value. For this reason, the only direct translation from CSP into `JCSP` is for communications of the form c, $c!exp$, or $c?x$. In [18], we provided a strategy to translate communication with multiple fields, but the restriction on communication values was maintained. Hence, communications like $c.0!x$ and $c.0?x$ were accepted, but the translation of complex communications was left as future work. Complex communications are defined as CSP prefixing in which at least one of the fields that is not the last one is a communication ($!exp$ or $?x$). For example, $c?x.0$ and c?x!0 are complex communications as both have an input at the first field.

Based on [18], *JCircus* [11] implements multiple-fields communications (*e.g.* $c.0.1$) as multi-dimensional arrays, in which each dimension represents a field of the communication. Our approach to accommodate the translation of complex communications in *JCircus* is to expand the possibilities of the communication. This expansion, however, may result in a very large (and possibly inefficient) code. For this reason, it is only applied to complex communications. For communications that were already supported by *JCircus*, we left the previous (and more efficient) approach.

By way of illustration, consider the **channel** $c : \{0,1\} \times \{0,1,2,3\}$. Based, on the channel's type declaration, we are able to infer that the action $c?x?y \rightarrow A$ is initially able to engage on eight possible synchronisation: from $c.0.0$ up to $c.1.3$. It is important to notice that both variables x and y are in the context of A and the chosen communication determines their values in A. Hence, after a given communication, we assign the corresponding values to the input variables. For that, we make use of an environment that maps communication possibilities to mappings variable-value. In our example, the environment is $\{0 \mapsto \{x \mapsto 0, y \mapsto 0\}, \ldots, 7 \mapsto \{x \mapsto 1, y \mapsto 3\}\}$.

For simple prefixing, it is clear what the following action is. In the example above, after a communication on c, the action behaves like A. Nevertheless, in external choices (possibly between two complex communications), we need to infer what the next action is based on the index of the choice. For example, let us consider the choice $c_1?x?y \rightarrow A_1(x,y) \Box c_2?x?y \rightarrow A_2(x,y)$. A further environment maps the index of the expanded choice to the index of the original choice. In this example, we have $\{0 \mapsto 0, \ldots, 7 \mapsto 0, 8 \mapsto 1, \ldots, 15 \mapsto 1\}$.

In Figure 3, we present a sketch of the generated code for this example. In Line `(03)`, we instantiate the object `ccMaps_1` that encapsulates both environments mentioned above. In Line `(22)`, we invoke the method `mapLeftOrRight`: we provide the chosen index of the expanded choice as argument to get the corresponding index in the original choice. For example, given `1 (c.0.1)`, the method returns `0` (left). In Lines `27` and `31`, we invoke the method `mapFromSelectToVarEnv` from `ccMaps_1` to get the correct values, given the index of the choice of the variables `x` and `y`, respectively. Two further auxiliary environments, one for each channel, allows a direct use of integers as array indexes. The method `abs` receives the channel name, the position of the communication and the value communicated and returns its corresponding absolute value (Lines `13-19`). In our example, this environment does not play an important role; it, however, is essential in the presence of negative numbers.

The translation of complex communication may lead to inefficient implementations causing scalability problems. The same problem applies for parts of the strategy that also requires an expansion of the communication possibilities like value communication in multi-synchronised channels (see Section 5.1). This problem, however, only happens if the channels involved have types with large cardinality. On the other hand, these strategies can be used in practice for non-problematic types such as free types, booleans, and small ranges of

```
(00) ...
(01) int select_1;
(02) /* Maps to index of choice and variable values */
(03) CCMaps_CCC_1 ccMaps_1 = new CCMaps_CCC_1 ();
(04)
(05) /* Mapping of original values to absolute values */
(06) Abs_c1 abs_c1 = new Abs_c1();
(07) Abs_c2 abs_c2 = new Abs_c2();
(08)
(09) Alternative a0 =
(10)  new Alternative (
(11)   new Guard []{
(12)    /* Original Index 0 */
(13)    c1 [abs_c1.abs ("c1", 0, new BigInteger ("0"))]
(14)    [abs_c1.abs ("c1", 1, new BigInteger ("0"))].getEnd(),
(15)    ...
(16)    /* Original Index 1*/
(17)    ...
(18)    c2 [abs_c2.abs ("c2", 0, new BigInteger ("1"))]
(19)    [abs_c2.abs ("c2", 1, new BigInteger ("3"))].getEnd()}
(20)   });
(21) select_1 = a0.select();
(22) switch (((Integer)ccMaps_1.mapLeftOrRight.get(select_1)).intValue()){
(23)  case 0:{
(24)   int x =
(25)    (Integer)
(26)    ((HashMap)
(27)      ccMaps_1.mapFromSelectToVarEnv.get (select_1)).get ("x");
(28)   int y =
(29)    (Integer)
(30)    ((HashMap)
(31)      ccMaps_1.mapFromSelectToVarEnv.get (select_1)).get ("y");
(32)   A1 (new CircusInteger (x), new CircusInteger (y));
(33)   break;
(34)  }
(35)  case 1:{
(36)   int x = ...
(37)   int y = ...
(38)   A2 (new CircusInteger (x), new CircusInteger (y));
(39)   break;
(40)  }
(41)}
```

Figure 3. Translating Complex Communications.

integers, which are vastly used in practice. Furthermore, our translation of complex communications as multi-dimensional arrays also allowed an integrated solution for simple point-to-point communications and multi-synchronised communications (either carrying values or not).

5.3. Sharing Parallel Composition and Interleaving

The parallel operator implemented in JCSP corresponds to Hoare's parallel composition, in which the processes synchronise on the events that they have in common. By way of illustration, let us consider the following processes:

$$\textbf{process}\, P \mathrel{\widehat=} \textbf{begin} \bullet a \rightarrow b \rightarrow d \rightarrow Skip\, \textbf{end}$$
$$\textbf{process}\, Q \mathrel{\widehat=} \textbf{begin} \bullet a \rightarrow c \rightarrow c \rightarrow Skip\, \textbf{end}$$

The only parallel composition, among various possibilities, that can be directly translated into JCSP `Parallel` is $P \,[\![\{\!| a |\!\}]\!]\, Q$. Parallel compositions like $P \,|||\, Q$ and $P \,[\![\{\!| a,b,c |\!\}]\!]\, Q$ cannot be directly translated using the `Parallel`. This is because the intersection between the alphabets of P and Q is $\{a\}$ and using `Parallel` they will synchronise exactly on $\{a\}$. As a consequence, *JCircus* did not support cases in which the processes shared channels that were not in the synchronisation channel set.

In this section, we describe a translation strategy that we have included into *JCircus*, which allows its users to use the full expressiveness of the *Circus* parallel composition and interleaving. The strategy is based on renaming and consists of the following steps:

1. Identification of the parallel branches;
2. Construction of a list of possible synchronisation for each channel. Each synchronisation is a set of branches' identifications and indicates that these branches must synchronise on the given channel;
3. Definition of the renaming for each channel of each branch of the parallelism;
4. Processing the renaming;
5. Hiding the renaming from the interface.

In the sections that follow, we present the details of each of these steps. We will use the action presented in Figure 4 to exemplify the strategy on each of the steps.

$$(a \rightarrow SKIP) \,[\![\{\!| a |\!\}]\!]\, \left(\begin{array}{l} (a \rightarrow SKIP) \\ ||| \\ \left(\left(\begin{array}{l} (a \rightarrow SKIP) \\ ||| \\ (a \rightarrow SKIP) \end{array} \right) [\![\{\!| a |\!\}]\!] \left(\begin{array}{l} (a \rightarrow SKIP) \\ ||| \\ (a \rightarrow SKIP) \end{array} \right) \right) \end{array} \right)$$

Figure 4. Expression to be used to explain the renaming strategy for the parallelism.

5.3.1. Step 1: Identification of the parallel branches

The branches of a parallel composition are all the leaves of the parallel expressions. For that, we consider only parallel operators (parallel composition and interleaving) as branching expressions and identify the leaves from left to right. By way of illustration, we present in Figure 5, the result of the branches identification of the action presented in Figure 4. In our example, all branches are prefixing actions $a \rightarrow Skip$.

The branches identification is used in an analysis of the parallel structure of the action that defines the synchronisation possibilities as we describe in the sequel.

5.3.2. Step 2: Construction of the synchronisation sets

The analysis of the parallel branches returns one list of synchronisation sets for each channel. Each synchronisation set of a channel represents a synchronisation possibility on that channel and contains the identification of all branches that take part on it. In our example, we have one

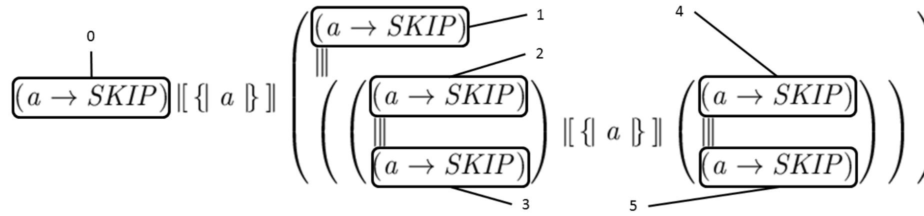


Figure 5. Branches and indexes of the expression of figure 5.

channel, *a*, that is mapped to its list of synchronisation sets. Our algorithm to build this list works recursively, for each channel, as follows: first, each leave composes a singleton set. We start from the left-most leaf $\langle\{0\}\rangle$. The algorithm then moves into the right-hand side of the top most parallel composition, whose left most leaves are $\langle\{2\}\rangle$ and $\langle\{3\}\rangle$. Moving upwards, these lists are not combined because *a* is not in the synchronisation channel set. Thus, the algorithm returns a simple concatenation of both lists, $\langle\{2\},\{3\}\rangle$. Similarly, the algorithm returns $\langle\{4\},\{5\}\rangle$ from the right-hand side of the parallel composition. Next, the parallel composition on *a* forces a combination of the results of the left-hand side and right-hand side: a cross-product between both list achieves this combination, $\langle\{2,4\},\{2,5\},\{3,4\},\{3,5\}\rangle$. Moving upwards in the parallel structure, the algorithm simply concatenates this list with the list of synchronisation from branch 1 because of the interleaving. The result of this concatenation is $\langle\{1\},\{2,4\},\{2,5\},\{3,4\},\{3,5\}\rangle$. Finally, at the top most parallel composition, the algorithm combines this list with the list $\{0\}$ that corresponds to the synchronisation at the left hand-side. The final result is $\langle\{0,1\},\{0,2,4\},\{0,2,5\},\{0,3,4\},\{0,3,5\}\rangle$.

5.3.3. Step 3: Defining the renaming of each parallel branch

The next step of the strategy is to define the renaming of each branch. For every channel *c*, we apply the renaming $c \leftarrow c_i$ (*c* becomes c_i) to the branch *b* if, and only if, *b* is a member of the *i*-th element in the the synchronisation list of *c*. For instance, the branch 0 is a member of every synchronisation set of channel *a*. For this reason, *c* is renamed to every new channel c_i. On the other hand, branch 1 is a member only of the first synchronisation set of *c*; we rename *c* only once. We present below all the renaming on *a* from our example.

- Branch 0: $[a \leftarrow a0, a \leftarrow a1, a \leftarrow a2, a \leftarrow a3, a \leftarrow a4]$
- Branch 1: $[a \leftarrow a0]$
- Branch 2: $[a \leftarrow a1, a \leftarrow a2]$
- Branch 3: $[a \leftarrow a3, a \leftarrow a4]$
- Branch 4: $[a \leftarrow a1, a \leftarrow a3]$
- Branch 5: $[a \leftarrow a2, a \leftarrow a4]$

The next step of the strategy is to apply the renaming to each of the corresponding nodes in the AST.

5.3.4. Step 4: Processing the renaming

The application of the renaming is relatively simple but behaves slightly different for functional renaming and non-functional renaming. The former renames channels to one new name only, whilst in the latter one channel is renamed to more than one new name. For instance, a functional renaming is applied to branch 1 and a non-functional renaming is applied to branch 0.

In cases where we have a functional renaming, the name of the channel is directly changed in the AST. Non-functional renaming, however, requires a slightly less simple change. In such cases, we replace simple prefixing $a \rightarrow A$ to an external choice on all new names followed by *A*. For example, branch 2 becomes $a1 \rightarrow Skip \;\Box\; a2 \rightarrow Skip$.

It is important to notice that the only change is on the channels' references, not in the communication structure. When the event to be renamed is a complex communication, for instance, the content of the communication fields is left unchanged. For example, if a communicates two values, the prefixing $a?x?y \rightarrow Skip[a \leftarrow a1, a \leftarrow a2]$ becomes $a1?x?y \rightarrow Skip \Box a2?x?y \rightarrow Skip$. Based on the strategy presented in Section 5.2, *JCircus* is able to translate this external choice. Multi-synchronisation is also supported. In this case, the branches that participate on a multi-synchronisation are members of the same synchronisation set. As a matter of fact, synchronisation sets with cardinality higher than 2 indicate a multi-synchronised channel.

The expression given in figure 5 is updated to the expression of figure 6. Internally, *JCircus* creates a basic process for each of the branches and replaces the branches by an invocation to its corresponding internal process.

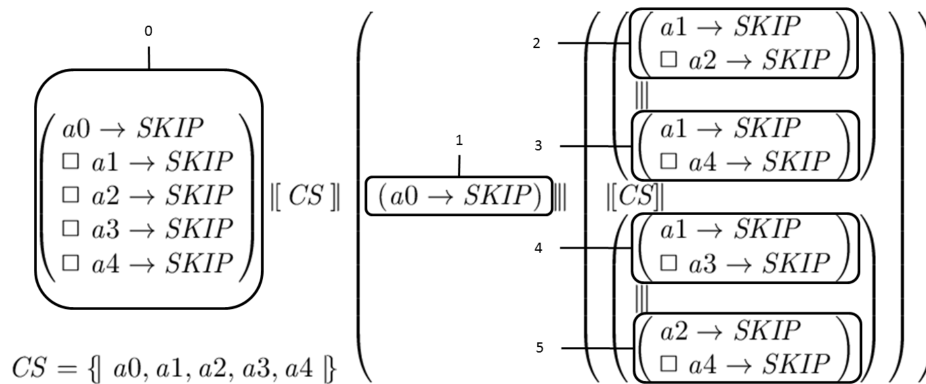


Figure 6. Updated branches and indexes of the expression of figure 5.

5.3.5. Step 6: Hiding the renaming from the interface

The external environment needs to be oblivious of the renaming used in our strategy. This is achieved by changing the behaviour of the GUI that is generated by *JCircus* as follows.

Every interaction of the environment with the generated system is done using buttons that are automatically generated by *JCircus*. Internally, the GUI propagates this interaction to the system by trying to synchronise on the channels that correspond to the environment's interaction in a random order. In our example, when the environment presses the a button, the GUI tries to synchronise on the channels $a0$ to $a4$, one by one, in a random order. For each of these channels, it waits for a certain time. If it timeouts, the next possibility is tried, and so on. If any of the attempts succeeds, the user receives a positive feedback. However, if none of them succeeds, nothing happens and the interaction is ignored. From the user perspective, the original channel is the only possibility of interaction. Nevertheless, internally in the GUI, this is replaced by a communication on any of the possible synchronisation on the new names given to that channel. This approach has been verified using FDR. The scripts can be found at the project webpage.

In Figure 7, we present a sketch of the generated code for the branch 0 in the example discussed above. As explained above, the renaming is applied to the branch before the translation. The result of the translation of a simple communication is a choice between the renamed instances of a ($a0$ up to $a4$) in corresponding branch. Afterwards, the action behaves according to the chosen event.

The current distribution of *JCircus* contains all the extensions discussed in this paper. Furthermore, we have integrated *JCircus* into CRefine, a tool that supports the *Circus* refinement calculus and its tactic language. This integration provides an integrated environment

```
int select_1;
select_1 = (
new Alternative (
    new Guard []{ a0.getEnd(), a1.getEnd(), a2.getEnd(),
                  a3.getEnd(), a4.getEnd()}
)).select();
switch (select_1) {
   case 0:{ (new Skip()).run(); break; }
   case 1:{ (new Skip()).run(); break; }
   case 2:{ (new Skip()).run(); break; }
   case 3:{ (new Skip()).run(); break; }
   case 4:{ (new Skip()).run(); break; }
};
```

Figure 7. Translating Alphabetised Parallel and Interleaving.

that allows users to formally develop systems in a calculational manner. In the next section, we discuss this integration.

5.4. *Integrating into* CRefine

CRefine [19] is a Java tool that automates the application of the *Circus* refinement calculus [18] and fosters a systematic correct-by-construction approach for systems developments. Using this approach, developers may start from a (usually centralised) abstract specification and, with a sequence of steps, achieve a (usually distributed) concrete specification. Each step is an application of a refinement law, which is usually valid under certain conditions that need to be proved. CRefine automatically manages the development and its proof obligations, most of which are automatically proved.

Sometimes, during the development using refinement calculus, the same laws are applied in the same manner in various developments or even in different parts of a single development. A strategy to optimize this calculus is to formalise these applications as refinement tactics, which can then be used as single transformation rules. Using this approach, the refinement calculus becomes more agile, reducing time and effort. CRefine's current distribution adds the support for the definition and application of refinement tactics to CRefine. This extension constitutes an useful addition that can be used while modelling systems in *Circus*. Using the new module, users can define and use tactics that considerably optimises the *Circus* development process.

The final result of this development process (using tactics or not), however, is not an executable program but a concrete specification which may be animated using tools like [17]. The integration of CRefine with *JCircus* allows a final translation of the concrete specification into a mainstream programming language. Using CRefine, the user simply clicks on the concrete specification and invokes the translation to Java. This integration provides *Circus* with a complete development path from abstract specification into code.

6. Performance analysis

In this section, we present an analysis of performance of the extensions presented in this paper. In these experiments, we used simple examples as input to *JCircus* varying the important parameters for each of the experiments, and collected the translation time, the execution time and the memory usage. The experiments allowed us to analyse the scalability of the approach and to identify issues which must be addressed in *JCircus*. In what follows, we

organise this section as follows. In Section 6.1, we describe the results of the experiments for multi-synchronisation. Next, Section 6.2 describes the results of the experiments for the translation of complex communications. Finally, in Section 6.3, we present the results of the experiment for the translation of sharing parallel composition and interleaving. A detailed account on the information provided in this Section can also be found at the project webpage.

6.1. Multi-Synchronisation Performance

The aim of this experiment was to compare the performance of the translation of multi-synchronisation using the previous solution (multi-synchronisation protocol) and the new one that uses `AltingBarriers`.

In this experiment, we used a specification in which a varying number of process take part in a given synchronisation. The experiment was executed using both versions of *JCircus*. The original version translated the process to a code that uses the multi-synchronisation protocol and the new version translated the process to a code that uses `AltingBarriers`. The overall results are presented in Figure 8.

Number of Processes	Translation (ms)		Execution (ms)		Memory (KB)	
	Protocol	Barriers	Protocol	Barriers	Protocol	Barriers
3	257	115	157,8	106,4	11276	14732
4	314	130	170,8	98	11232	14568
5	254	143	152,8	97,8	11208	14520
30	1764	344	307,3	136,6	11228	15636
100	7052	17949	548,6	378	11692	19680

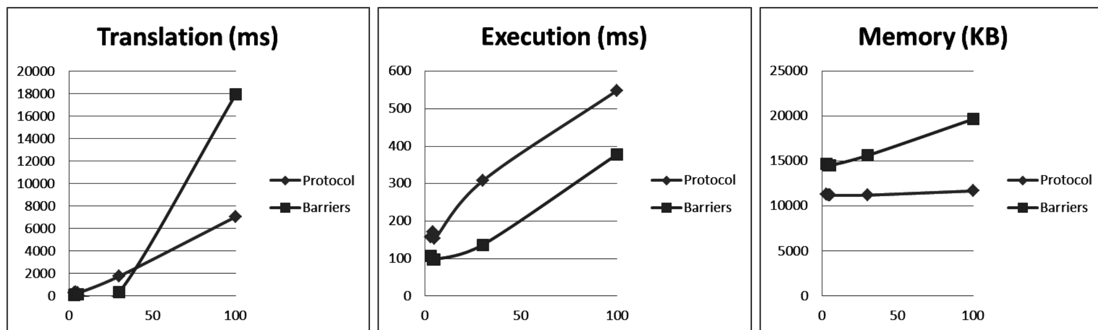


Figure 8. Experiments Results: Multi-Synchronisation.

The translation time was smaller for the solution we implemented using alting barriers. However, this was only valid for an experiment with up to around 35 processes taking part in the multi-synchronisation. After this number, the translation using the multi-synchronisation protocol becomes more efficient. This is due to the nature of their growth rate: the `AltingBarrier` solution has an exponential growth rate; initially it has a relatively small growth rate which becomes rather large as we increase the number of participants. On the other hand, the use of the protocol presented a linear growth rate. This showed us that the user must be given a choice on which solution *JCircus* should use. This choice is being currently included in the tool.

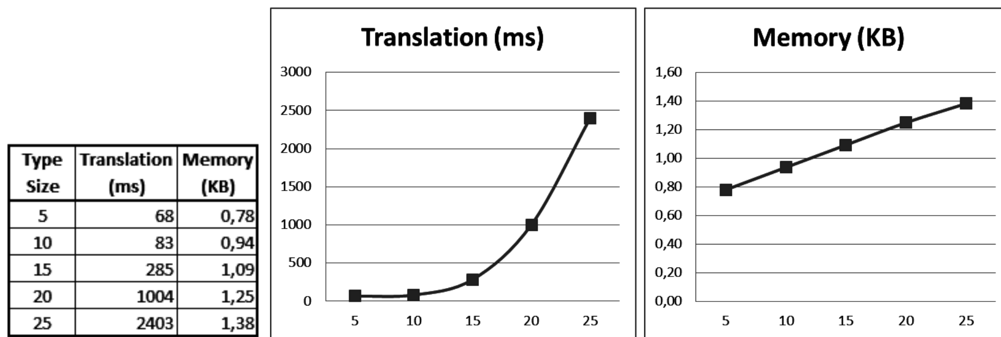
The results of the experiment demonstrate that the execution time of the multi-synchronisation protocol is higher than the execution time of the solution using barriers. This difference happens because the protocol executes various point-to-point simple communications between a controller that manages the processes that take part in the synchronisation in a two-phase commitment manner. On the other hand, `JCSP AltingBarrier` uses a fast implementation that is not a two-phase commit. It has overheads that are linear with respect to the number of barrier offers being made [25]. On the other hand, the memory experiments demonstrated a higher memory usage for the multi-synchronisation using `AltingBarrier`.

Both increasing rates, however, are linear with respect to the number of participant processes. Hence, the execution of the barriers approach proved to be feasible even in a network of one hundred processes.

6.2. Complex Communications Performance

The experiment on the translation results of complex communications was done using a simple example of one communication between a reader and a writer on a single **channel** c. The experiment was divided into two parts. In both parts, the execution time presented a very low variation (below 10ms) and we omit it in the sequel.

In the first part, we fixed the number of communication fields in 2 (**channel** $c : T \times T$) and increased the size of the channel type T from 5 to 25 elements. The overall results of this first part are presented in Figure 9. The growth in the memory usage was not crucial because its growth proved to be linear. Nevertheless, as expected, the experiments demonstrated that the growth in the translation time has an exponential rate. It, however, was still considerably manageable for types of size below 30. These results shown us a important limitation in the use of complex communications: these should only be used for channels whose types have a relatively small cardinality. For example, using such complex communication in the presence of integers (within the Java boundaries) is not viable.



Type Size	Translation (ms)	Memory (KB)
5	68	0,78
10	83	0,94
15	285	1,09
20	1004	1,25
25	2403	1,38

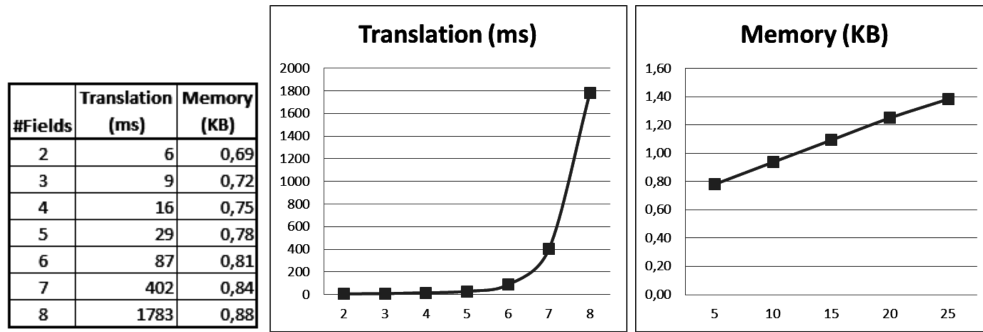
Figure 9. Experiments Results: Complex Communication (increasing type size).

In the second part, we fixed the size of type T in 2 and increased the number n of fields in the communications on c from 2 to 8 (**channel** $c : T \times ... \times T$ - n times). The overall results of this second part are presented in Figure 10. The growth in the memory usage also proved to be linear, hence, not a problem for scaling. The experiments shown an exponential growth of the translation time on the number of fields in the communication. It was, however, still considerably manageable for up to 8 communication fields. Although apparently a limitation, this does not constitute a problem given that to the best of our knowledge on the practical use of CSP, communications with over 8 communication fields are not largely used (if used at all).

6.3. Sharing Parallel Composition and Interleaving Performance

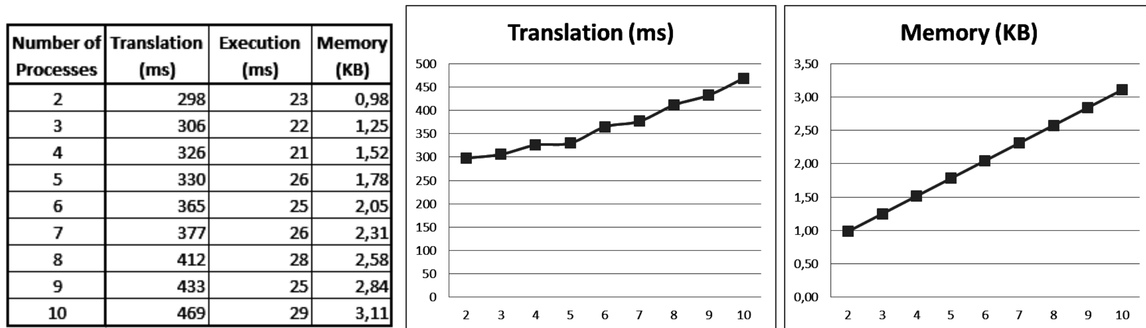
The last experiment investigated the translation of sharing parallel composition and interleaving. In this experiment we considered a simple example on which one process is willing to synchronise on a channel end and the other end is shared among various processes forcing an interleaving on that end to occur. Hence, synchronisation happens between two process at a time only. The results of this experiment are presented in Figure 11, in which we increase the number of readers from 2 up to 10.

The translation of sharing parallel composition and interleaving is strongly based on previous solutions. Nevertheless, this is only done after a considerable change to the AST that



#Fields	Translation (ms)	Memory (KB)
2	6	0,69
3	9	0,72
4	16	0,75
5	29	0,78
6	87	0,81
7	402	0,84
8	1783	0,88

Figure 10. Experiments Results: Complex Communication (increasing communication size).



Number of Processes	Translation (ms)	Execution (ms)	Memory (KB)
2	298	23	0,98
3	306	22	1,25
4	326	21	1,52
5	330	26	1,78
6	365	25	2,05
7	377	26	2,31
8	412	28	2,58
9	433	25	2,84
10	469	29	3,11

Figure 11. Experiments Results: Sharing Parallel Composition and Interleaving.

reflects the necessary renaming. For this reason, the linear growth in the memory consumption was as expected. On the other hand, the experiment also indicated a linear growth rate in the translation time.

7. Conclusions

In this paper, we proposed an extension to *JCircus*, an automatic translator from *Circus* to Java. We extended *JCircus* by providing: (1) a new optimised translation strategy to multi-way synchronisation; (2) the translation of complex communications, and; (3) the translation of CSP sharing parallel and interleaving. The extensions are presented in Table 1. We also described the integration of *JCircus* into the *Circus* refinement tool, CRefine, which provided *Circus* with a tool support for the whole development process. Finally, a performance analysis of the resulting code was also presented and discussed.

Multi-synchronisation was originally implemented using a multi-synchronisation two-phase commitment protocol. However, from December 2006, `JCSP` provided the class `AltingBarrier`, which implements multi-synchronised channels that can engage in an external choice. This implementation optimises the two-phase commitment protocol because `JCSP AltingBarrier` uses a fast implementation whose overheads are linear with respect to the number of barrier offers being made [25]. Our experiments presented in Section 6.1 demonstrates this optimisation. They, however, also indicate that a trade-off must be made between the execution time and the translation time. The translation using `AltingBarrier` was quicker for networks of up to around 35 processes. Because of the exponential nature of the time growth of this translation, the translation using the protocol, whose growth rate was linear, proved to be faster. This indicated a further improvement to *JCircus* that is being currently implemented: the user must be given a choice on which solution *JCircus* should use.

The extension presented in this paper also provided *JCircus* with the possibility to translate complex communication, in which at least one of the fields that is not the last one is a

Feature	**Example**	**1.0**	**2.0**
no actions	*Stop*	✓	✓
successful termination	*Skip*	✓	✓
chaotic behaviour	*Chaos*		
simple prefix	$c \to A$	✓	✓
output prefix	$c!x \to A$	✓	✓
input prefix	$c?x \to A$ **restrictions**	✓ no nesting like using same variable $c?x \to c?x \to A$	✓ none
restricted input prefix	$c?x?P \to A$		
complex prefix	$c?x!y \to A$		✓
sequential composition	$A_1 ; A_2$	✓	✓
external choice	$A_1 \Box A_2$ **restrictions**	✓ only prefixing actions no output event	✓ only prefixing **and guarded** actions
internal choice	$A_1 \sqcap A_2$	✓	✓
parallel composition	$A_1 \lbrack\lbrack ns_1 \mid cs \mid cs_2 \rbrack\rbrack A_2$ **restrictions**	✓ multi-sync with single writer multi-sync with same processes $\alpha(A_1) \cap \alpha(A_2) \subseteq cs$	✓ none
interleaving	$A_1 \lbrack\lbrack ns_1 \mid\mid cs_2 \rbrack\rbrack A_2$ **restrictions**	✓ $\alpha(A_1) \cap \alpha(A_2) = \emptyset$	✓ none
hiding	$A \setminus cs$		
renaming	$P[a \leftarrow b]$		
parametrisation	$A(x)$	✓	✓
recursion	$\mu X \bullet A(x)$	✓	✓
assignment	$x := v$	✓	✓
alternation	$\mathbf{if} g_1 \to A_1 \rrbracket g_2 \to A_2 \ \mathbf{fi}$	✓	✓
variable block	$\mathbf{var} x : T \bullet A$	✓	✓
deadlock-free GUI			✓

Table 1. *Circus* Actions Translation.

communication ($!exp$ or $?x$). For example, $c?x.0$ and c?x!0 are complex communications as both have an input at the first field. This limitation was due to the absence of an algorithm to generate the possibilities of communication for a given channel. In this paper, we presented an approach to implement such algorithm that allows the translation of communications with an arbitrary number of field decorations. This approach used mappings that stored information about the chosen communication in order to determine the values of the input field variables. The experiments presented here indicated that this approach has limitations due to the exponential growth in the translation time as we increase the size of the types involved in complex communications. It, however, was still considerably manageable for types of size below 30. An interesting piece of future work is to investigate other translation strategies for complex communications that do not require the expansion of the communication possibilities.

The translation of parallel composition in the previous version of *JCircus* required the intersection of the alphabets of the processes to be a subset of the synchronisation channel

set. For this reason, interleaving on channels that were in the intersection of the processes' alphabets was not possible. In this paper, we remove this restriction by providing a strategy for translating *Circus* parallel composition with any synchronisation channel set. This required the investigation of an approach to force the interleaving between two channels that would, otherwise, synchronise. We presented a strategy that internally renames the channels based on the parallel structure of the main process. The environment remains oblivious of the renaming; hence, the strategy's implementation is equivalent to the specification and its correctness has been validated using FDR.

The extensions described in this paper can also be used in the translation of further languages into JCSP. This is due to the fact that here we focus on the translation strategy rather than only on the *Circus* to JCSP strategy. For instance, the translation strategy for sharing parallelism and interleaving provides a correspondence between Hoare's parallel composition and sharing parallel composition. Thus, any parallel composition from an event-based language that uses sharing parallel composition can be translated into another event-based language that uses Roscoe's parallel composition like OCCAM.

Despite the large set of translatable *Circus* constructs, there are still further possible extensions to *JCircus* that can be done. An interesting challenge is the translation of hiding, which is not trivial because its operational behaviour depends on the position of the hidden events and may introduce divergence, deadlock, and non-determinism.

Another approach to extend *JCircus* is to provide some of the constructors natively in JCSP, using more efficient algorithms in Java. A native availability of the constructors in JCSP would make the translation process more direct and efficient removing scalability problems of the translation process like that presented in Section 6.2. Furthermore, as in the case of multi-synchronisation, the efficiency of the generated code would also be considerably better.

At last, *JCircus* was integrated into CRefine, a tool that supports the *Circus* refinement calculus and tactic language. This integration allows a final translation of the calculated concrete specification into a mainstream programming language and provides *Circus* with a complete development path from abstract specification into code.

Acknowledgments

Angela Freitas has originally worked on *JCircus* and helped to solve some issues related to the tool's extension. Leo Freitas has provided some insights related to the CZT. INES and CNPq partially supports the work of Marcel Oliveira: grants 573964/2008-4 and 560014/2010-4. The COMPASS project (FP7- ICT Call 7 IP) also collaborated with the work presented here.

References

[1] Community Z Tools. At http://czt.sourceforge.net/.

[2] D. Carrington, D. Duke, R. Duke, P. King, G. A. Rose, and G. Smith. Object-Z: an object-oriented extension to Z. In Son T. Vuong, editor, *Formal Description Techniques, II (FORTE'89)*, pages 281–296, 1990.

[3] 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.

[4] ClearSy. Atelier-B. http://www.atelierb.eu/en/.

[5] David Crocker. Perfect developer: A tool for object-oriented formal specification and refinement. tools exhibition notes at formal methods europe. In *In Tools Exhibition Notes at Formal Methods Europe*, page 2003, 2003.

[6] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[7] M. S. C. Filho and M. V. M. Oliveira. Extending CRefine to Support Tactics of Refinement. In A. Simão and C. Morgan, editors, *14th Brazilian Symposium on Formal Methods - Short Papers*, pages 55 – 60. ICMC/USP, September 2011.

[8] C. Fischer. CSP-OZ: A combination of Object-Z and CSP. In H. Bowman and J. Derrick, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS'97)*, volume **2**, pages 423–438. Chapman & Hall, 1997.

[9] C. Fischer. How to combine Z with a process algebra. In J. P. Bowen, A. Fett, and M. G. Hinchey, editors, *ZUM'98: The Z Formal Specification Notation, 11th International Conference of Z Users*, number 1493 in Lecture Notes in Computer Science, pages 5–23. Springer Verlag, 1998.

[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, Dec 2005.

[11] A. Freitas and A. L. C. Cavalcanti. Automatic Translation from *Circus* to Java. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *14th International Symposium on Formal Methods*, volume **4085** of *Lecture Notes in Computer Science*, pages 115–130. Springer, Aug 2006.

[12] L. Freitas. *Model-checking Circus*. PhD thesis, Department of Computer Science, The University of York, 2005. YCST-2005/11.

[13] A. Galloway. *Integrated Formal Methods with Richer Methodological Profiles for the Development of Multi-perspective Systems*. PhD thesis, School of Computing and Mathematics, University of Teeside, 1996.

[14] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[15] M. G. Hinchey and S. A. Jarvis. *Concurrent Systems: Formal Development in CSP*. McGraw-Hill, Inc., New York, NY, USA, 1995.

[16] B. P. Mahony and J. S. Dong. Blending Object-Z and Timed CSP: an Introduction to TCOZ. In K. Torii, K. Futatsugi, and R. A. Kemmerer, editors, *The 20th International Conference on Software Engineering (ICSE'98)*, pages 95–104. IEEE Computer Society Press, 1998.

[17] D. Oliveira and M. V. M. Oliveira. Joker: An Animation Framework for Formal Specications. In A. Simão and C. Morgan, editors, *14th Brazilian Symposium on Formal Methods - Short Papers*, pages 43 – 48. ICMC/USP, September 2011.

[18] M. V. M. Oliveira. *Formal Derivation of State-Rich Reactive Programs using Circus*. PhD thesis, Department of Computer Science, University of York, 2006.

[19] M. V. M. Oliveira, A. C. Gurgel, and C. G. de Castro. CRefine: Support for the *Circus* Refinement Calculus. In Antonio Cerone and Stefan Gruner, editors, *6th IEEE International Conferences on Software Engineering and Formal Methods*, pages 281–290. IEEE Computer Society Press, 2008.

[20] M. V. M. Oliveira and J. C. P. Woodcock. Automatic Generation of Verified Concurrent Hardware. In M. M. Larrondo-Petrie M. Butler, M. Hinchey, editor, *9th International Conference on Formal Engineering Methods*, volume 4789 of *Lecture Notes in Computer Science*, pages 286 – 306. Springer-Verlag, November 2007.

[21] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall Series in Computer Science. Prentice-Hall, 1998.

[22] H. Treharne and S. Schneider. Using a process algebra to control B operations. In K. Araki, A. Galloway, and K. Taguchi, editors, *Proceedings of the 1st International Conference on Integrated Formal Methods*, pages 437–456. Springer, June 1999.

[23] P. Welch, N. Brown, J. Moores, K. Chalmers, and B. Sputh. Alting barriers: synchronisation with choice in Java using JCSP. *Concurr. Comput. : Pract. Exper.*, 22(8):1049–1062, June 2010.

[24] P. H. Welch. Process oriented design for Java: concurrency for all. In H. R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 51–57. CSREA Press, June 2000.

[25] P. H. Welch, N. Brown, J. Moores, K. Chalmers, and B. H. C. Sputh. Integrating and Extending JCSP. In A. McEwan, S. A. Schneider, W. Ifill, and P. H. Welch, editors, *CPA*, volume 65 of *Concurrent Systems Engineering Series*, pages 349–370. IOS Press, 2007.

[26] P. H. Welch, G. S. Stiles, G. H. Hilderink, and A. P. Bakkers. CSP for Java: multithreading for all. In B. M. Cook, editor, *Architectures, Languages and Techniques for Concurrent Systems*, volume **57** of *Concurrent Systems Engineering Series*. IOS Press, April 1999.

[27] J. C. P. Woodcock and A. L. C. Cavalcanti. A concurrent language for refinement. In A. Butterfield and C. Pahl, editors, *IWFM'01: 5th Irish Workshop in Formal Methods*, BCS Electronic Workshops in Computing, Dublin, Ireland, July 2001.

[28] J. C. P. Woodcock and J. Davies. *Using Z—Specification, Refinement, and Proof*. Prentice-Hall, 1996.