Linking a “State-rich” Process Algebra to a “State-free” Process Algebra to Verify Software / Hardware Implementations

Arshad Beg
Lero@TCD
School of Computer Science & Statistics
Trinity College Dublin
Rep. of Ireland
begm@scss.tcd.ie

Andrew Butterfield
Lero@TCD
School of Computer Science & Statistics
Trinity College Dublin
Rep. of Ireland
Andrew.Butterfield@scss.tcd.ie

ABSTRACT

Following the development of formalisms based on data and behavioural aspects of the system, there are number of attempts in which these two formalisms are mixed together to get benefit of both paradigms. ‘Circus’ being a living specification language with continuous collaboration from both academia and industry, is a combination of Z, CSP and the refinement calculus. To make use of the available and industry-proven tools for a particular programming paradigm, there is a need to develop a formally verified link between the one world and the other. The aim of this work is to develop a formally verified link between a state-rich process algebra i.e. ‘Circus’ to a state-free process algebra i.e. CSP. To achieve the research goal, the most suitable available tools are to identify. For developing link between targeted formal languages, we will identify the key translations required between the two languages. For ensuring correctness of the translation, we will formalise the key translation / refinement steps. These will form the theoretical core of the work and support the soundness of the link. In the end, we will select and verify a case study from the collection of software / hardware protocols.

1. MOTIVATION

1.1 Research Objective

The aim of this work is to develop a formally verified link between ‘Circus’-based tool to a CSP based tool. Tools in the link are to support the verification of software / hardware implementations which are derived from Circus specifications. The motivation behind developing such links is to glue together the formal methods tools i.e. model checkers, theorem provers and model-animations etc to run in an automatic or semi-automatic cyclic manner, with each tool working in the right place. After developing link between the tools, the objective will be to formally verify all the translation strategy of the link by relating the semantics of Circus and CSP within the UTP framework. The need of developing link between these two tools is due to the fact the model-checker for ‘Circus’ is still under development while there is an academically and industry-proven model-checker for ‘CSP’.

1.2 Area of Proposed Contribution

‘Formal Methods’ can be defined as a collection of languages, techniques and tools based on mathematics for specifying and verifying systems\(^1\). More precisely, ‘Formal Methods’ is about ‘Formalising’ a system on a basis of a set of tools and notations having a formal semantics. These tools are used to clearly specify the requirements of a system, allowing the proof of properties of that specification and to prove the correctness of an implementation with respect to that specification. The key motivation for developing the link is to contribute to the ‘Grand Challenge in Computing’ (GC6) project. The GC project[13] is expected:

\begin{itemize}
  \item 1. to deliver a comprehensive and verified theory of programming
  \item 2. to give a prototype for a comprehensive and integrated suite of programming tools
  \item 3. to deliver a repository of verified software
\end{itemize}

So, the development of the link will be a contribution to the prototype collection of integrated suite of programming tools.

History of Circus: In 2000, the development of Circus was started by University of York in collaboration with Universidade Federal de Pernambuco, Brazil. Since then there have been continuous collaboration by British Industry as well as academia in Ireland, Macau and Brazil. Circus is a living language and has an active research focus from the researchers as the Circus website shows that there are more

\(^1\)http : //en.wikipedia.org/wiki/Formal\_methods
than 50 publications since 2001. These publications deals with the underlying theory of the language as well as its practical use in industrial applications.

1.3 Technical (or Research) Approach

1. Identify the most suitable tools for use. Identify the key translations required between the two process algebras.
2. Formalise key translation / refinement steps. These will form the theoretical core of the work and support the soundness of the link between the two process algebras.
3. Apply to a wider range of case studies which can be from hardware-based protocols or networking protocols.

2. BACKGROUND READING

2.1 The Z notation, CSP and Circus

The Z language [24] has its basis in the set theory and mathematical logic. Standard set operators, set comprehensions, Cartesian products, and power sets are used in the set theory. First-order predicate calculus is used in the mathematical logic. In Z, the mathematics can be structured by the use of schemas. Collection of mathematical objects and their properties is described using these schemas. A unique type is given to each mathematical object in the language. So, the use of types gives the functionality of checking the type of each object in the specification. The use of refinement is another characteristic feature of the Z language. A model is defined using simple mathematical data types to describe the desired behaviour. Then this description is refined by construction of another model. This refined model gives a description closer to the implementation of the system. Similar kind of notations are B-method and Object-Z (a Z extension with an object oriented approach).

Communicating Sequential Processes (CSP) [12] is a process algebra where the systems are represented as processes. In the CSP world, the system is specified as the order in which these processes are to be carried out. CSP allows concurrency and provides a way in which these processes can interact. This is achieved through channels. Messages can be exchanged between these channels from one process to another one. However, data requirements are not very well dealt in the process algebras. In CSP, parameters can be defined for a limited data requirement. The Calculus for Communicating Systems (CCS) [20] is another example of a process algebra.

2.1.1 An Example of a program in CSP

In this example, the door of a lift can be opened or closed, as exemplified by the type DoorState and there are four possible events in which the lift may engage i.e. up, down, open and close.

datatype DoorState = opened | closed
channel up
channel down
channel open
channel close

INITIAL_LIFT = LIFT(0,closed)

LIFT(floor,doorState) = (floor < 5) & (up -> LIFT(floor + 1, doorState)) ]]
(floor > 0) & (down -> LIFT(floor - 1, doorState)) ]]
(doorState == closed) & (open -> LIFT(floor,opened)) ]]
(doorState == opened) & (close -> LIFT(floor,closed)) ]]

There are number of efforts in which the process algebras and state-rich formal languages are mixed together to get benefits from both paradigms. This is necessary for the systems which have both data and behavioural requirements. Some of these examples are CSP|B [23], Z with CSP [21], Z with CCS [10] and many more. ‘Circus’ is a specification language that combines Z, CSP and refinement calculus constructs. The main difference between Circus and other ones is that the languages i.e. Z and CSP are mixed freely in a specification. A Circus program consists of a sequence of paragraphs. These paragraphs can be a: Z paragraph, channel or channel set definition, or a process declaration. A process paragraph may contain a Z paragraph, an action definition or a name set definition. The process paragraph is started and ended with the keywords begin and end.

Example of a Specification in Circus.

DoorState ::= opened | closed

processLift, up, down, open, close

state

LiftState ⊑ begin

floor : N

doorState : DoorState

InitLift ⊑ (floor := 0; doorState := closed)

Lift ⊑ (floor < 5) \uparrow \uparrow floor := floor + 1

{ floor > 0 & down \uparrow floor := floor - 1 }

doorState = closed & open \rightarrow (doorState := opened)

{ doorState = opened & close \rightarrow (doorState := closed) }

• InitLift; μX • (Lift; X)

end

If the examples of CSP and Circus are analysed critically, there are apparent differences between two among which few are discussed here. In the lift example of CSP, the process INITIAL_LIFT is initiated with the call to process LIFT by passing parameters to it with some initial values e.g. LIFT(0, closed). While in the Circus example, the state variables defined by the Z schema LiftState are initialized with assignment statements in the process InitLift. Furthermore, the same difference appears in the guarded commands of actions in a Circus specification where after making a decision about a state of the variable e.g. floor < 5, the state variables changes its state being assigned an expression e.g. floor + 1. While, in the CSP world, the same operation is implemented by passing parameters e.g. LIFT(floor + 1, doorState).

While mixing two different languages, the unification of the semantics of the languages is a matter of concern. So, there
must be a unification framework so that two worlds of these languages could be mixed together.

2.2 Unifying Theory of Programming

The Unifying Theories of Programming (UTP) [11] proposes a unification of different programming paradigms based on the theory of relations. The unification allows the exploitation of different paradigms. The relation between the paradigms can result in mappings that relate specifications in abstract models to programs in more concrete models; in UTP the refinement relation is simply a logical implication. The semantics of Circus in UTP framework are explained in detail in [22]. Furthermore, the UTP semantics of CSP are discussed in [6].

2.3 Introduction to the Available Tools

Community Z Tools - CZT: CZT is an open-source Java framework to build formal method tools for Z and Z dialects. The specifications in LaTeX, Unicode and XML formats can be parsed, typechecked, transformed, animated and printed using the formal method tools included in CZT. The supported languages of the latest version of CZT are Z, Object Z and Circus. A limited subset of Z is supported by the animator available in the tool.

Failure Divergence Refinement - FDR: FDR [8] is a CSP [12] based model-checking tool. More precisely, FDR can be described as a refinement checker. Refinement is a term for the process of incremental implementation of the system from the specification. In general, usually it is not possible to construct a program directly from its specification, then prove it to be correct. Instead, the program should be constructed in small steps, each time adding more detail. Since the changes are small, it is relatively easy to prove at each stage that the implementation satisfies the specification. If S is a specification and P is a program then P ⊑ S means that the program P refines the specification S. The FDR tool does refinement checking based on the traces, failures and failures/divergence models.

Saoithin: Saoithin [1] is a theorem prover having its design based on the Unifying Theories of Programming (UTP) framework. It is based on the UTP literature [11] so that it can support the proofs containing higher order logic, alphabets and “programs as predicates”. It mainly deals proofs in equational style.

2.4 Possible Case Study Area

The description of background readings in the following subsection belongs to the possible case study area of my PhD work. There are number of other readings but are not mentioned here due to space limitation.

2.4.1 Verification of Flash Memory Behaviour

After the start of ‘Grand Challenge in Computing’ (GC6) [13] project with a special focus on mission critical filestores, a number of efforts have been made to formalise the flash memory and filestores. Paper [4] gives the Z notation for the formal model of NAND flash memory. The model describes the internal architecture of NAND flash memory with some abstractions. Paper [2] is a step ahead towards mechanising the formal model of NAND flash memory. The Z/Eves Theorem Prover has been used for describing the state model and initialisation operation of NAND flash memory. Papers [5, 3] are about modelling the flash memory behaviour using CSP. In these works, Open NAND Flash Interface (ONFi) specifications are modelled. Instead of writing CSP directly, the ONFi’s finite state machines’ specifications are converted into intermediate form using State Chart XML (SC–XML). This XML was then automatically converted into CSP via XML Transforms (XSLT).

Paper [19] reports on the use of Alloy (a model checker) and HOL (a theorem prover) to validate and verify a VDM model of the Intel Flash File System Core specification, as a part of the ‘Verifiable File System’ (VFS) project. Paper [14] describes the formal modeling and analysis of a design for flash-based filesystem in Alloy. The authors modelled the basic operations of filesystem as well as other features that are crucial to NAND flash hardware, such as wear-leveling and erase-unit reclamation. Papers [16, 15, 17, 18] document experiments in the formal verification of OneNAND Flash Memory which is a trademark of Samsung Electronics.

3. CURRENT WORK

3.1 Converting Circus to CSP

To establish the link between Circus based tools and CSP based tool, the specifications written in one domain are to be translated in another domain. It has already been mentioned that CZT tools contains the parser, typechecker and printer for Circus. The figure 1 explains the basic concept behind the tool inter-operability of CZT and FDR.

If we could get the translation from the CZT tool which contains the typechecker and parser of Circus to a format which is readable by the FDR, we would be able to develop a link for Circus specifications. In the latest version of the freely available CZT tool sources, no one ever had an attempt to make CZT and FDR work together which is a clear indication that it would be a first-ever work. In the technical report [9], the translation from CSP to FDR work is described but not the other way.

In the previous work [7], a transformation strategy for transforming from a concrete Circus specification to a Java pro-

![Figure 1: Translation from Circus to CSP](http://czt.sourceforge.net/manual.html)
gram has been proposed. It consists of translation rules that, applied to each Circus construct in a concrete specification, result in a Java program that implements the Circus program. The resulting Java program uses the JCSP library, a Java implementation of the CSP model for concurrency and communication. The work [7] provides an implementation of the translation strategy. The implementation result is a tool called JCircus. This tool generates a Java implementation of the concrete Circus specification through a very simple GUI. The work presented in this paper is the starting point for translating from Circus to CSPM, as this provided us the basis for parsing Circus specifications. The next step will be to modify the translation rules written for conversion of Java, to work with conversion to CSPM.

The basic architecture of JCircus is shown in the figure 2. JCircus contains three main modules. The parser is the first one, which receives a LaTeX file containing the specification, parses it, and creates the AST that represents the specification. The AST is given as input to the type checker, which performs type inference, checks for type errors, and annotates the AST nodes for expressions with their types. The third module is the translator which is the contribution of [7]. The TranslatorVisitor class contains all the methods to convert the circus processes to their java equivalent.

Figure 2: The basic architecture of JCircus Classes. Figure 4.4, [7]

3.2 Simple Example of a Translation Rule

Rule for Internal Choice In case of a java program, the rule for internal choice is to use RandomGenerator class to produce a pseudo-random number, which is compared in a switch statement to pick a random process.

```java
int chosen = RandomGenerator.generateNumber(1,n);
switch(chosen) {
    case 1: {
        Proc1
    }
    ...
    case n: {
        Proc
    }
}
```

In case of CSPM, the rule will be varied as follows:

\[
\left[[\text{Proc}_1 \cap \ldots \cap \text{Proc}_n]\right]_{\text{Proc}} = \left[[\text{Proc}_1]\right]_{\text{Proc}} \upharpoonright \ldots \upharpoonright \left[[\text{Proc}_n]\right]_{\text{Proc}}
\]

This rule means that in the case of translation from Circus to CSPM, Circus non-deterministic choice operator is simply replaced with the CSPM equivalent operator i.e. \(\upharpoonright\).

In order to formally translate each constructor of Circus into CSPM, the semantic justification of the translation has to be established. As discussed earlier, we have the opportunity of already having semantics of both Circus and CSP in UTP. So, the main objective of the research will be to formalise all the translation strategy rules by relating the semantics of Circus and CSP within the UTP framework.

Let us take an example of a simple process as shown in the figure 3.2. In case of Circus process, \(x\) is assigned value in the specification. While for doing same in CSPM, the value is passed as a parameter. In case of this particular example, we need to extend UTP semantics to cover parameteric processes.

4. SUMMARY OF ACCOMPLISHMENTS AND REMAINING WORK

The key accomplishments so far are:

1. The raw initial idea described in this paper was presented to the panel of international reviewers in November 2009 at the Lero workshop at Athlone, Ireland.
2. Updated Flash Memory Behaviour model [5, 3] discussed in section 2.4.1 from ONFi 1.0 to ONFi 2.1. This work is accepted in a conference. This work provided me basis for working with FDR toolkit.
3. Identification of the key tools (CZT and FDR).
4. Identification of the Java framework for converting the concrete Circus specifications into Java i.e. JCircus.
5. REFERENCES


