Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 195 (2008) 75–93
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Type Checking *Circus* Specifications

Manuela Xavier[1],[2]

Centro de Informática
Universidade Federal de Pernambuco
Recife, Brazil

Ana Cavalcanti[3]

Department of Computing Science
University of York
York, United Kingdom

Augusto Sampaio[4]

Centro de Informática
Universidade Federal de Pernambuco
Recife, Brazil

Abstract

Circus is a formal language that combines Z, CSP and additional constructors of Morgan's refinement calculus. It is aimed at the development by refinement of state-rich reactive systems. In this work, we define the *Circus* type system and describe the design and implementation of a type checker. We developed the type checker based directly on the typing rules that formalise the type system of *Circus*. We believe that this contributed to the robust construction of the type checker. We also discuss the validation strategy of the type checker, including integrations with other *Circus* tools.

Keywords: Formal methods, concurrency, formal languages, type system, refinement tool.

1 Introduction

The increasing need for the development of systems with quality is resulting in industrial interest in the application of formal techniques. Currently, there are formal languages for modeling complex data structures, like Z [21,25], and others for

[1] Thanks to Angela Freitas and Leo Freitas for their suggestions of improvements to the implementation of the type checker and tests. The work of Manuela Xavier was partly supported by CNPq. Ana Cavalcanti is partly supported by QinetiQ and the Royal Society, England.

[2] Email: manuela.xavier@gmail.com

[3] Email: Ana.Cavalcanti@cs.york.ac.uk

[4] Email: acas@cin.ufpe.br

doi:10.1016/j.entcs.2007.08.027

modeling communication and concurrency, like CSP [9]. *Circus* [23,24] is a specification, design and programming language that combines Z and CSP. Additionally, *Circus* includes constructors of Morgan's refinement calculus [14] and Dijkstra's guarded commands [3]. The *Circus* semantics is based on Hoare and He's Unifying Theories of Programming [8].

The integration of a language for data modeling with an algebra of processes makes possible the adequate modeling of a wider range of systems. There are other works that consider the integration of Z, or one of its extensions, with some algebra of processes [4,5,20,12,10,22]. Differently from these languages, *Circus* includes a refinement calculus [2] that allows us to get code from an abstract specification.

At the moment, a number of tools that automate several aspects of the use of *Circus* are under development: a model checker [7], a theorem prover [17], and a refinement editor [26] are a few examples. These tools need a type checker to provide guarantees about the type consistency of the specifications and programs, and of their results.

In this paper, we present a formal definition of the *Circus* type system, with the intention of assisting the development of a type checker for this language. At first, we define the *Circus* type rules, and afterwards we describe how we implemented the type checker using a direct mapping of the type rules to code. This contributed to the robustness of the type checker. The type checker offers extra facilities, such as the annotation of type information for each fragment of a specification or program, and the availability of clear and objective error messages. Additionally, we designed the type checker as a component of easy integration, maintenance and evolution. In order to validate the type checker in practice, we have integrated it into *JCircus* (a tool that translates *Circus* to Java). We have also developed a new tool, CircusRefine, which supports the applications of refinement laws to *Circus* specifications, using the type checker to ensure type consistency.

In Section 2, we describe some aspects of the *Circus* syntax. We present a formal definition of the *Circus* type system in Section 3. Following this, in Section 4, we present the type checker. We discuss our validation strategy in Section 5. Finally, in Section 6 we discuss our results and describe some future works.

2 *Circus*

A *Circus* program is composed of a sequence of paragraphs, including Z paragraphs, channel definitions, channel set definitions, and process definitions. In Figure 1, we can see a *Circus* specification with some channels and one process called *CookieMachine*. This machine receives money and returns one cookie and the change, if necessary.

In *Circus*, a process declaration is composed of the keyword **process**, followed by the name of the process and its definition. A process can be defined in terms of other processes, using the operators of sequential composing ($P_1;\ P_2$), internal choice ($P_1 \sqcap P_2$), external choice ($P_1 \Box P_2$), parallelism ($P_1 \parallel P_2$), interleaving ($P_1 \interleave P_2$), etc.

$COOKIE ::= ok \mid notok$

$\textbf{channel}\ in, change : \mathbb{N}$

$\textbf{channel}\ out : COOKIE$

$\textbf{process}\ CookieMachine \mathrel{\widehat=} \textbf{begin}$

$\quad | \ \ cookieValue, cookieQuantity : \mathbb{N}$

$\quad \textbf{state}\ State \mathrel{\widehat=} [\, money, quantity : \mathbb{N}\,]$

$\quad StateInit \mathrel{\widehat=} [\, State' \mid money' = 0 \land quantity' = cookieQuantity\,]$

$\quad InputMoney \mathrel{\widehat=} [\, \Delta State;\ x? : \mathbb{N} \mid money < cookieValue \land money' = money + x?\,]$

$\quad Input \mathrel{\widehat=} money < cookieValue \ \&\ in?x \rightarrow InputMoney$

$\quad OutputCookie$

$\quad \Delta State$

$\quad o! : COOKIE$

$\quad money \geq cookieValue$

$\quad ((quantity > 0 \land money' = money - cookieValue \land quantity' = quantity - 1 \land o! = ok)$

$\quad \lor$

$\quad (quantity = 0 \land money' = money \land o! = notok))$

$\quad Output \mathrel{\widehat=} money \geq cookieValue \ \&$

$\qquad\qquad (\textbf{var}\ o : COOKIE \bullet OutputCookie;\ out!o \rightarrow change!money \rightarrow Skip)$

$\quad \bullet\ StateInit;\ (\mu\ X \bullet (Input \mathbin{\Box} Output);\ X)$

$\textbf{end}$

Fig. 1. *CookieMachine* Specification

For the definition of a basic process, we use the keywords **begin** and **end**. A definition is composed of a list of process paragraphs, which can be empty; a definition of state, introduced by the keyword **state**; and a main action that defines the behavior of the process. In *Circus*, we can use Z paragraphs to define the state of a process. In our example, we define *CookieMachine* as a basic process with two state components: *money* and *quantity*.

A process paragraph can be Z paragraphs, name set definitions or action definitions. A Z paragraph allows the definition of constants and types which are local to a process. In our *CookieMachine* specification, *cookieValue* and *cookieQuantity* are defined as local constants. Additionally, we can use Z paragraphs to define process actions; a Z paragraph is an action if it defines operations on the process state. This action can change the state, but it does not communicate values. In our example, *OutputCookie* is an action defined by a Z schema. It captures the effect on the state of a cookie purchase. *StateInit* and *InputMoney* are also examples of schema. The former initiates the state of the *CookieMachine* process, and the latter describes the effect of money deposit in the machine.

We can also define an action in *Circus* in the form: $N \mathrel{\widehat=} act$, where N is the action name, and act describes the behavior of the action. *Input* is an example of such an action. Its definition combines CSP and Z constructs. As in CSP, *Circus* also has the prefixing operator and the guard constructor, but only for actions (not for composing processes). These constructors are generally used together to define actions, as we can observe in *Input*. The action defined as $p \ \&\ act$ behaves like

act only if the predicate p is true; otherwise, it is blocked. On the other hand, an action defined like $c?x \rightarrow act$ allows the input of values through the channel c and, afterwards, behaves like act; the received value is stored in x. To communicate output values, the communication has to be: $c!e \rightarrow act$. In this case, the expression e must have the same type of the c channel. The values communicated by the channel are called communication parameters; they can be input parameters $(?cp)$ or output parameters $(!cp$ or $.cp)$, and can be combined arbitrarily in a single prefixing.

Additionally, *Circus* allows the recursive definition of actions through the μ operator. A recursive action $\mu X \bullet act$ uses the name X to make recursive calls; the name X is used in act as an action name. The main action of the $CookieMachine$ specification is recursive; first of all, it calls to $StateInit$ action, and then repeatedly offers the choice of input $(Input$ action) and output $(Output$ action).

We can define parametrised or indexed processes. Process indexing is one of the *Circus* operators that is not defined in CSP. An indexed process, for example $i : T \odot P$, has the same behavior as P, but uses different channels, defined implicitly. If P uses a channel $in : \mathbb{N}$, the indexed process communicates values through the channel $c\_i : T \times \mathbb{N}$, that is implicitly defined. This channel communicates pairs, where the type of the first element is always the index type, and the type of the element is always the type of the original channel.

For more details on *Circus*, including examples, refer to [2,16,26].

3 Type System for *Circus*

As *Circus* is a combination of Z and CSP languages, it is natural to think of combining the rules of their type systems [29,11] to characterise the type system of *Circus*. However, the Z and the CSP constructs are freely mixed within *Circus*, and it also includes Dijkstra's constructs of guarded commands. In addition, we are not aware of a widely accepted formal definition for the CSP type system. The CSP type rules presented in [11] are incomplete, because the focus of that work is only the process calculus and it does not consider constructs such as channel declarations. Additionally, as far as we are aware, there are no implementations of CSP type checkers that adopt this type system. On the other hand, a widely used CSP type checker developed by Formal Systems Europe Limited covers all the CSP syntax, and, even though it is not associated with a formal definition of a type system, it is our reference for typing in CSP. In the case of Z, the formal definition of its type system [29] is complete and well established, but is difficult to read.

The type system that we define here is based on many aspects of the Z and the CSP type systems, with the intention to keep *Circus* compatibility with these languages. We do not redefine the rules of Z. Instead, we reference these rules and define rules for the additional constructs of *Circus*. We decided to use a particular notation that is simpler than the one used by the Z type system, as we present in the following sections.

3.1 Type Environment

Before describing the type rules, we define some basic elements that are not within the syntax of *Circus*: environments, which represent the context of a term in consideration. The type rules define a relation written $\Gamma \rhd A : T$ to mean that A has type T in the environment Γ.

Data types T are the types of channels, constants and variables. They can be primitive types (as $\mathbb{N}$, $\mathbb{Z}$, standing for naturals and integers), names of given sets N, or a symbol that represents an absence of type (wt). The last one is the type of a synchronized channel.

$$T \in \mathit{Typ} ::= N \mid \mathbb{A} \mid \mathbb{N} \mid \mathbb{Z} \mid wt \mid ...\mathit{other\ types}$$

Besides data types T, other term types θ are available for programs, *Circus* paragraphs, channel declarations, processes, actions, etc.

$$\theta ::= T \mid \mathbf{Program} \mid \mathbf{CircusParagraph} \mid \mathbf{Process} \mid \mathbf{Action} \mid ...$$

We call the set of well-formed type environments of *Circus* *TEnv*. A *Circus* type environment is a register with twenty fields. Amongst them, one part is responsible for keeping global definitions, and another part is responsible for keeping definitions that are local to a process.

Some of the fields that record external definitions to processes are: *channels*, that maps all channel names to types; *processes*, that stores all process names of a program; *definedProcs*, that stores the names of processes that have already been defined; *zDefs*, that maps all names of definitions extracted from Z paragraphs (global constants, free types, given types and generic types) to their types; and *defNames*, which joins the available information of other fields (like the names of channels, name sets, processes, global constants and types) and is used to simplify definitions.

Some of the fields related to internal definitions to processes are: *localVars*, that stores all local variables in scope and the state components; *actions*, that stores all action declarations; *definedActs*, that stores the names of actions that have already been defined; *localZDefs*, that maps the names and types extracted from Z paragraphs; and *localDefNames*, that stores all names of local definitions and is used to simplify definitions.

There are some constraints (invariants) on the values these fields can take. The *defNames* field includes the names of the *definedProcs* field, together with the names in the domain of the *channels* and *zDefs* fields.

$$\mathit{defNames} = \mathit{definedProcs} \;\cup\; \mathrm{dom}\, \mathit{channels} \;\cup\; \mathrm{dom}\, \mathit{zDefs}$$

On the other hand, the *localDefNames* field registers only local definitions. It includes the names of *definedActs* fields, and the names in the domain of the *localVars* and *localZDefs* fields.

$$\mathit{localDefNames} = \mathit{definedActs} \;\cup\; \mathrm{dom}\, \mathit{localVars} \;\cup\; \mathrm{dom}\, \mathit{localZDefs}$$

All names of the defined processes are in the set of process names ($definedProcs \subseteq processes$). In the same way, all names of the defined actions are in the set of action names ($definedActs \subseteq actions$).

3.2 Type Rules

Type rules validate judgments, possibly based on judgments already known as valid. Judgments have the form $\Gamma \rhd \Im$, where $\Im$ is an assertion as, for example, $A : T$. Each type rule contains a possibly empty set of premises above a horizontal line, and a single conclusion below the line. The premises of the rules can be other judgments, as well as restrictions. When all premises are satisfied, the conclusion is valid.

In the following sections, we define the *Circus* type rules for some of its syntactic categories. We illustrate some rules using the *CookieMachine* example in Figure 1. The complete list of the *Circus* type rules can be found in [26].

3.3 Program

A *Circus* program is well typed if all its paragraphs are well typed; see Table 1. In the beginning of a *Circus* program verification, the type environment is empty. We use the symbol $\Gamma_\emptyset$ to represent an empty type environment in which all fields do not contain any information. The function *ExtractProcDefs* receives as argument a list of *Circus* paragraphs, and gives as result an environment that contains all the process names extracted from the list; the formal definition of *ExtractProcDefs* and of all other functions used in the type rules presented here can be found in [26]. The list *cpl* of *Circus* paragraphs that compose a program has to be well typed in the context of *ExtractProcDefs cpl*. This requirement allows a process to be referenced before being defined, and so, allows the definition of mutually recursive processes. In our example, the *CookieMachine* process name is included in the environment, more specifically in the *processes* field.

The type rules for paragraph lists appear in Table 2. A list of *Circus* paragraphs is well typed if the first paragraph (cp) is well typed and the remaining list of paragraphs (cpl) is well typed in an environment enriched by the global definitions of cp. The function *Defs* receives as arguments a *Circus* paragraph and the current environment, and returns another extended environment that contains all global definitions of the paragraph. The $\oplus$ operator is environment overriding. In this way, the verification of each *Circus* paragraph is done with an environment updated by the definitions of the previous paragraphs. As an example, in the *CookieMachine* specification, as the *COOKIE* given type is defined in the beginning, the subsequent *Circus* paragraphs can use it.

$$\frac{(ExtractProcDefs\ \ cpl) \rhd cpl : \mathbf{CircusParagraphList}}{\Gamma_\emptyset \rhd cpl : \mathbf{Program}}$$

Table 1
Typing of *Circus* Program

$$\frac{\Gamma \rhd cp : \mathbf{CircusParagraph}}{\Gamma \rhd cp : \mathbf{CircusParagraphList}} \qquad \frac{\begin{array}{c}\Gamma \rhd cp : \mathbf{CircusParagraph} \\ (\Gamma \oplus (Defs\ cp\ \Gamma)) \rhd cpl : \mathbf{CircusParagraphList}\end{array}}{\Gamma \rhd cp\ cpl : \mathbf{CircusParagraphList}}$$

$$\frac{\begin{array}{c}(NewDefs\ p\ \Gamma.defNames) \\ \Gamma \rhd p : \mathbf{ZParagraph}\end{array}}{\Gamma \rhd p : \mathbf{CircusParagraph}} \qquad \frac{\Gamma \rhd cd : \mathbf{CDeclaration}}{\Gamma \rhd \mathbf{channel}\ cd : \mathbf{CircusParagraph}}$$

$$\frac{\begin{array}{c}\Gamma \rhd ln : \mathbf{CDeclaration} \\ \Gamma \rhd e : \mathbf{Expression}(\mathbb{P}\ T)\end{array}}{\Gamma \rhd (ln : e) : \mathbf{CDeclaration}} \qquad \frac{\begin{array}{c}\Gamma \rhd cd_1 : \mathbf{CDeclaration} \\ (\Gamma \oplus Chan\ cd_1) \rhd cd_2 : \mathbf{CDeclaration}\end{array}}{\Gamma \rhd cd_1;\ cd_2 : \mathbf{CDeclaration}}$$

Table 2
Typing of *Circus* Paragraphs

3.4 *Circus Paragraphs*

A Z paragraph, in *Circus*, is well typed if it is well typed as defined by the Z type rules, and if it does not define global names that already have been defined before as processes, channels, channel sets, global constants or types. The condition *NewDefs* (see Table 2) is true if, and only if, the names defined in the paragraph are new. We present the association between Z and *Circus* type rules in Section 3.10.

The paragraphs that declare channels and processes are well typed if the involved declaration (CDeclaration or ProcessDeclaration, respectively) is well typed. We omit the trivial rules for conciseness.

3.5 *Channels*

A declaration of channel with type is well typed if the channel names are new and distinct, and if the expression that represents the channel type is a well typed set. The channel declarations in the *CookieMachine* specification are well typed because the channel names are distinct and new in the global context.

A composition of channel declarations is well typed if the first declaration is well typed and the second is well typed in an environment updated by the channel names of the previous declaration. The application of the function *Chan* to a channel declaration returns an environment that contains all the channel names associated with their respective types. This allows us to avoid repeated channel declarations.

3.6 *Processes*

As defined in Table 3, a process declaration is well typed if its name has not been declared and if the process definition is well typed.

The function *FindImplicitChans* receives as arguments a process definition (p) and the current type environment (Γ). It defines a set that contains all the implicit channels extracted from the process definition. Implicit channels arise from indexed process definitions. These channels are defined from each channel used by the process. If, for example, an indexed process is defined with an index $i : T_1$ and if this process uses a channel $c : T_2$, the channel c_i is defined implicitly and its type is determined by the index and channel type $(T_1 \times T_2)$.

The condition *DeclareNewChans* applied to the set of implicit channels (returned by the function *FindImplicitChans*) and to the current type environment requires that the names of the extracted implicit channels are new or channels with the same type indicated by the index.

3.7 Basic Process

In the case of a basic process, it is well typed if the state, the list of paragraphs and the main action are well typed too, as defined in Table 4.

To determine whether a basic process is well typed, it is necessary to extract all the action names defined within the process so that mutually recursive actions are properly handled. The function *ExtractActDefs*, applied to a list of paragraphs, defines the environment with all the action names introduced in the paragraphs. With the overriding operator, the initial environment is extended and, as result, we have the environment Γ_1. The definitions of the actions, name sets, local types and constants declared before the definition of the state are captured in the environment Γ_2. The function *DefsPL* receives as arguments a list of process paragraphs and the current type environment, and returns another environment (Γ') with all definitions of the list. The function *DefsState*, when applied to the state paragraph, yields an environment (Γ'') that has only state components. The environment Γ_2 defines the context of the state paragraph. The environment Γ_3 includes information about:

$$\frac{\begin{array}{c} n \notin \Gamma.defNames \qquad \Gamma \rhd p : \mathbf{ProcessDefinition} \\ (DeclareNewChans\ (FindImplicitChans\ p\ \Gamma)\ \Gamma) \end{array}}{\Gamma \rhd \mathbf{process}\ n \mathrel{\widehat{=}} p : \mathbf{ProcessDeclaration}} \qquad \frac{\Gamma \rhd p : \mathbf{Process}}{\Gamma \rhd p : \mathbf{ProcessDefinition}}$$

Table 3
Typing of Process Declaration

$$\frac{\begin{array}{c} \Gamma_1 \rhd pl_1 : \mathbf{PParagraphList} \qquad \Gamma_2 \rhd sc : \mathbf{StateParagraph}(d) \\ \Gamma_3 \rhd pl_2 : \mathbf{PParagraphList} \qquad \Gamma_4 \rhd a : \mathbf{Action} \\ NotRedeclare((\Gamma_1.localDefNames), (\Gamma'.localDefNames)) \\ NotRedeclare((\Gamma_2.localDefNames), (\Gamma''.localDefNames)) \\ NotRedeclare((\Gamma_3.localDefNames), (\Gamma'''.localDefNames)) \end{array}}{\Gamma \rhd \mathbf{begin}\ pl_1\ \mathbf{state}\ sc\ pl_2\ \bullet\ a\ \mathbf{end} : \mathbf{Process}}$$

$$\begin{array}{c} \mathbf{where}\ \Gamma_1 = \Gamma \oplus (ExtractActDefs\ pl_1) \oplus (ExtractActDefs\ pl_2), \Gamma_2 = \Gamma_1 \oplus \Gamma',\ \Gamma_3 = \Gamma_2 \oplus \Gamma'', \\ \Gamma_4 = \Gamma_3 \oplus \Gamma''',\ \Gamma' = (DefsPL\ pl_1\ \Gamma_1),\ \Gamma'' = (DefsState\ sc\ d),\ \Gamma''' = (DefsPL\ pl_2\ \Gamma_3) \end{array}$$

$$\frac{\begin{array}{c} \Gamma \rhd p : \mathbf{PParagraph} \qquad (\Gamma \oplus \Gamma') \rhd pl : \mathbf{PParagraphList} \\ NotRedeclare((\Gamma.localDefNames), (\Gamma'.localDefNames)) \end{array}}{\Gamma \rhd p\ pl : \mathbf{PParagraphList}}$$

$$\mathbf{where}\ \Gamma' = (DefsP\ p\ \Gamma)$$

Table 4
Typing of a Basic Process

$$\frac{\begin{array}{c} n \in (\Gamma.\mathit{actions}) \\ n \notin (\Gamma.\mathit{parActions}) \end{array}}{\Gamma \triangleright n : \mathbf{Action}} \qquad \frac{\Gamma \triangleright c : \mathbf{ParCommand}}{\Gamma \triangleright c : \mathbf{Action}} \qquad \frac{\begin{array}{c} \Gamma \triangleright c : \mathbf{Communication} \\ (\Gamma \oplus \mathit{VarsC}\ \ c\ \Gamma) \triangleright a : \mathbf{Action} \end{array}}{\Gamma \triangleright c \rightarrow a : \mathbf{Action}}$$

$$\frac{\begin{array}{c} \Gamma \triangleright p : \mathbf{Predicate} \\ \Gamma \triangleright a : \mathbf{Action} \end{array}}{\Gamma \triangleright p \ \&\ a : \mathbf{Action}} \qquad \frac{\begin{array}{c} \Gamma \triangleright a_1 : \mathbf{Action} \\ \Gamma \triangleright a_2 : \mathbf{Action} \end{array}}{\Gamma \triangleright a_1 \ \Box\ a_2 : \mathbf{Action}} \qquad \frac{(\Gamma \oplus (\mathit{definedActs} = \{n\})) \triangleright a : \mathbf{Action}}{\Gamma \triangleright \ \mu\, n \bullet a : \mathbf{Action}}$$

Table 5
Typing of Actions

the definitions extracted from the first paragraph list, the state components, and the action names of the process. Finally, the environment Γ_4 contains all information in Γ_3 plus additional information about actions, name sets, local types and constants declared after the definition of the state. This environment defines the context for the type verification of the main action. The condition *NotRedeclare* requires that there are no repeated names with different types in the local definitions of two environments. For example, $\mathit{NotRedeclare}(\Gamma_1.\mathit{localDefsNames}, \Gamma'.\mathit{localDefNames})$ guarantees that the local names of Γ' have not been defined in the environment Γ_1 as local names.

The verification of a list of process paragraphs requires the type verification of each paragraph. In this case, the environment of each paragraph is enriched by the definitions of previous paragraphs. The function *DefsP* receives as arguments a process paragraph and the current environment, and returns another extended environment that contains all local definitions of the paragraph. Additionally, it is verified if the names declared in a process paragraph have not been already declared into the process.

3.8 Actions

A process paragraph can be an action definition and it is well typed if the action name has not been already declared within the process, and if the action definition is well typed too (rule omitted). In Table 5 we present some action rules.

An action reference is well typed if the cited name was declared before as an action and it is not a parametrised action name. In the *CookieMachine* example, the references to *InputMoney*, *OutputCookie*, *Input* and *Output* are well typed because they are names of actions and they were not declared as parametrised actions.

In the case of a recursive action $\mu\ X \bullet A$, it is well typed if A is well typed in an environment enriched with X as an action name. The context of this new name is only the action involved in the recursion. In the *CookieMachine* example, the context of the name X is only the sequential composition $(\mathit{Input} \ \Box\ \mathit{Output});\ X$

A prefixing is well typed if the communication is well typed and if the action is well typed in an environment extended by the input variables of the communication. The types of the input variables are determined by the type of the channel used in the communication. The function *VarsC*, applied to the communication and the current type environment, returns another environment that contains all input variables associated with the type extracted from the channel type. This environment also contains the variations of the extracted input variables. To explain the extraction of input variables, we consider the following program.

$$
\begin{array}{l}
[T] \\[1ex]
\textbf{channel}\, c : \mathbb{N} \times T \times \mathbb{P}\,\mathbb{N} \\
\textbf{process}\, P \mathrel{\widehat=} \textbf{begin} \\[1ex]
\qquad A \mathrel{\widehat=} c?x?y?z \rightarrow \mathit{Skip} \\[2ex]
\qquad B \mathrel{\widehat=} c?v?w \rightarrow \mathit{Stop} \\[2ex]
\qquad C \mathrel{\widehat=} c?t \rightarrow \mathit{Skip} \\
\bullet\; A;\; B;\; C\ \textbf{end}
\end{array}
$$

The input variables extracted from the communication of the action A are: $x : \mathbb{N}$, $x' : \mathbb{N}$, $x? : \mathbb{N}$, $x! : \mathbb{N}$, $y : T$, $y' : T$, $y? : T$, $y! : T$, $z : \mathbb{P}\,\mathbb{N}$, $z' : \mathbb{P}\,\mathbb{N}$, $z? : \mathbb{P}\,\mathbb{N}$ and $z! : \mathbb{P}\,\mathbb{N}$. The variable x, as it is the first parameter of the communication, has the type of the first component type of the channel, in this case, $\mathbb{N}$. The same occurs with the parameter y, but in this case, it has the type of the second component type of the channel. Finally, as z is the last parameter, it has the remaining type of the channel. In the communication of the action B, the input variables extracted are: $v : \mathbb{N}$, $w : T \times \mathbb{P}\,\mathbb{N}$ and their variations. As we have only two parameters of communication, the type of the last parameter is the tuple $T \times \mathbb{P}\,\mathbb{N}$. Finally, in the communication of the action C, as the channel c is communicating only one parameter, it has the complete type of the channel. Therefore, the input variables extracted are: $t : \mathbb{N} \times T \times \mathbb{P}\,\mathbb{N}$, $t' : \mathbb{N} \times T \times \mathbb{P}\,\mathbb{N}$, $t? : \mathbb{N} \times T \times \mathbb{P}\,\mathbb{N}$ and $t! : \mathbb{N} \times T \times \mathbb{P}\,\mathbb{N}$.

3.9 Communication

A communication that involves value passing is well typed if the channel has been declared to have a type, and if each one of the communication parameters is well typed. The number of parameters has to be less than or equal to the number of component types that compose the channel type. The function *NumTypesChan* receives as argument the type of the channel involved in the communication, and returns the quantity of component types. A type has more than one component type when it has a form of a cartesian product. Considering the example of the previous section, as the channel c has the type $\mathbb{N} \times T \times \mathbb{P}\,\mathbb{N}$, the function *NumTypesChan* will yield 3, as it has three component types: $\mathbb{N}$, T and $\mathbb{P}\,\mathbb{N}$. However, if a channel has the type $\mathbb{P}(\mathbb{N} \times \mathbb{N})$, the function returns 1, since it has only one component type.

The function *Extract*, in turn, receives as arguments a list of parameters and the channel type, and returns a set that contains all input variables of the communication and their variations, as we have explained in the previous section. In turn, this set is passed as argument to the function *NoConflicts* and it returns true if and only if it does not have multiple declarations of the same name with different types.

A list of communication parameters is well typed if each involved communication parameter is well typed. The environment of a communication parameter is updated by the input variables (and its variations) extracted from the previous communication parameters. The functions *GetTypeHead* and *GetTypeTail* receive as argument the channel type. The former yields the first component type of the channel type, and the latter returns a type without the first component type of the channel type. For example, if a channel type is $\mathbb{P}(\mathbb{N} \times T) \times U \times \mathbb{P}\,\mathbb{N}$, the function *GetTypeHead* returns the type $\mathbb{P}(\mathbb{N} \times T)$, and the function *GetTypeTail* yields the type $U \times \mathbb{P}\,\mathbb{N}$.

The function *ExtractVarsCP*, in turn, receives as arguments a communication parameter, the type of this parameter, and returns an environment with all input variables and its variations. The environment Γ is updated, through the overriding function, with these input variables, and allows the verification of the remaining parameters of the communication with these variables in scope, as we present in the example below.

$\mathbf{channel}\ c : \mathbb{N} \times \mathbb{N} \times \mathbb{N}$

$\mathbf{process}\ P \mathrel{\widehat{=}} \mathbf{begin}$

$\quad ...$

$\quad A \mathrel{\widehat{=}} c?x?y!x + y \rightarrow \mathit{Stop}$

$\bullet\ A\ \mathbf{end}$

In the action A, the expression of the third communication parameter $(x + y)$ uses the input variables extracted from the previous parameters. It is only possible due to the context extension of the input variables of the communication parameters.

$$\frac{\begin{array}{c} n \in \mathrm{dom}(\Gamma.\mathit{channels}) \\ ((\Gamma.\mathit{channels}\ n) \neq \mathit{wt}) \qquad \sharp\mathit{cpl} \leq (\mathit{NumTypesChan}\ (\Gamma.\mathit{channels}\ n)) \\ (\mathit{NoConflicts}\ (\mathit{Extract}\ \mathit{cpl}\ T)) \qquad \Gamma \rhd \mathit{cpl} : \mathbf{CParameterList}(T) \end{array}}{\Gamma \rhd n\ \mathit{cpl} : \mathbf{Communication}}$$

$$\mathbf{where}\ T = \Gamma.\mathit{channels}\ n$$

$$\frac{\begin{array}{c} \Gamma \rhd \mathit{cp} : \mathbf{CParameter}(\mathit{GetTypeHead}\ T) \\ (\Gamma \oplus (\mathit{ExtractVarsCP}\ \mathit{cp}\ T)) \rhd \mathit{cpl} : \mathbf{CParameterList}(\mathit{GetTypeTail}\ T) \end{array}}{\Gamma \rhd \mathit{cp}\ \mathit{cpl} : \mathbf{CParameterList}(T)}$$

$$\frac{\Gamma \rhd \mathit{cp} : \mathbf{CParameter}(T)}{\Gamma \rhd \mathit{cp} : \mathbf{CParameterList}(T)} \qquad \frac{}{\Gamma \rhd ?\mathit{cp} : \mathbf{CParameter}(T)} \qquad \frac{\begin{array}{c} T' == T \\ \Gamma \rhd e : \mathbf{Expression}(T') \end{array}}{\Gamma \rhd !e : \mathbf{CParameter}(T)}$$

Table 6
Typing of Communication

$$\frac{\Sigma \vdash^{\mathcal{D}} p \fatsemi \sigma}{\Gamma \rhd p : \textbf{ZParagraph}} \qquad \frac{\Sigma \vdash^{\mathcal{E}} sc \fatsemi \tau}{\Gamma \rhd sc : \textbf{Schema-Exp}} \qquad \frac{\Sigma \vdash^{\mathcal{P}} p}{\Gamma \rhd p : \textbf{Predicate}}$$

$$\textbf{where } \Sigma = (\mathit{CircusToZTypeEnv}\ \Gamma)$$

Table 7
Association between Z and *Circus* Type Rules

An input parameter is always well typed. The name of the input variable is new (we have a new context) and its type is extracted from the channel type. On the other hand, an output parameter is well typed if the involved expression is well typed and it has the type of the channel or of the component corresponding to the expression in the compound type channel.

3.10 Association between Z and Circus Type Rules

The validation of Z paragraphs, expressions, predicates and declarations in *Circus* is made through references to the Z type rules. However, to associate the rules of both systems, we need to map the *Circus* type environment to the Z type environment.

In the formal definition of the Z type system [29], *TypeEnv* is the set of Z type environments. These environments associate names to types. Also, Σ denotes a type environment ($\Sigma : \mathit{TypeEnv}$).

We determine how a *Circus* type environment can be used to define a Z type environment. For this, we define a function that receives as argument a *Circus* type environment (Γ) and returns the Z type environment (Σ).

$$\begin{array}{l} \mathit{CircusToZTypeEnv} : \mathit{TEnv} \pfun \mathit{TypeEnv} \\ \mathit{CircusToZTypeEnv}\ \Gamma = \Gamma.\mathit{zDefs} \cup \Gamma.\mathit{localZDefs} \end{array}$$

The resulting type environment includes all global and local definitions extracted from Z paragraphs. These definitions are stored in the *zDefs* and *localZDefs* fields of the *Circus* type environment Γ, respectively.

The format of the Z type rule is similar to the format that we adopt in this work. The judgments of paragraphs, expressions, predicates and declarations have the form: $\Sigma \vdash \Im$, where $\Im$ is an assertion. Nevertheless, Z adopts different assertions for each syntactic structure. The type rules that we present in Table 7 establish the association between Z and *Circus* type rules. In the premise of the rules we have descriptions like Z judgments, and in the conclusion we have judgements on *Circus* types.

A Z paragraph in *Circus* is well typed if in the context of the type environment defined by Σ, the paragraph p has a signature σ as defined in the Z type rule; a signature consists of a function that maps names to types. A schema expression in *Circus* is a Z expression. Then, it is well typed if in the context of the type environmet defined by Σ, the schema expression sc has a type τ. Finally, a predi-

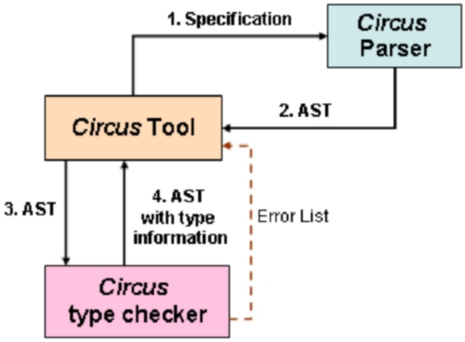

Fig. 2. Integration between a *Circus* Tool and the Type Checker

cate in *Circus* is well typed if it is well typed in accordance with the Z type rules. Consequently, in the context of the type environment Σ, the predicate p is well typed.

In total, there are 118 rules in the formalisation of the *Circus* type system. They are captured in the implementation of the type checker that we present in the next section.

4 Type Checker for *Circus*

We can use the type rules to verify programs and guarantee the consistent use of values and expressions in *Circus* specifications. However, as in many formal techniques, the application of the type rules is an arduous work that can result in errors. Moreover, type verification is an activity that can be automated, even for rich languages like *Circus*, that involves specification constructs. In fact, the type system of a language can be designed specifically to make automatic verification of types possible.

The development of a *Circus* type checker demands the implementation of a Z type checker. Our idea was to select one of the Z type checkers based on the ISO standard [29] and extend it. This prevents rework, but it is important to choose a robust and well designed Z type checker.

We can find many tools that automate the utilization of Z. However, a disorganized development of Z tools can result in a situation where we have a lot of tools that are useful but are not compatible. The CZT - Community Z Tools - is a project that has as goal to tackle this problem [15,13]. Its intention is to offer an open source framework, developed in Java, which allows an easy construction of tools for Z and its extensions like Object-Z [19], TCOZ [12], and currently, *Circus*.

The CZT framework has a well designed, robust and continuously tested type checker for standard Z. The design of the CZT type checker is based on the visitor design pattern, and this is a good solution for the development of this kind of tool because it is a structure that allows an easy navigation within the syntax of a program specification. We chose this tool as a starting point for our work.

Our type checker was developed as a component, with an interface for integra-

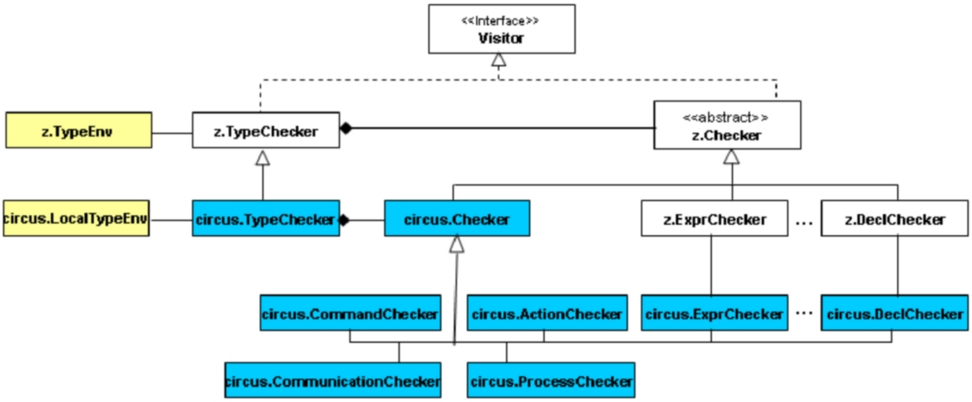

Fig. 3. Classes of the *Circus* Type Checker

tions with other *Circus* tools. Figure 2 shows how the integration can be carried out.

To perform a type verification, any *Circus* tool has to use the parser to generate a *Circus* AST (Abstract Syntactic Tree), which must be passed on to the type checker. If the specification does not have type errors, the type checker returns `true` and annotates the original AST with type information. If the specification has some type error, the type checker returns `false` and a list of error messages describing the type errors is made available. Independently of the result, the *Circus* tool may or may not continue its processing.

4.1 Architecture

Figure 3 illustrates the architecture of the *Circus* type checker, built as a substantial extension of the Z type checker of the CZT. The *Circus* classes `TypeChecker` and `Checker` inherit from corresponding classes of the Z package: `z.TypeChecker` and `z.Checker`. This allows *Circus* classes to have the same functionality of Z classes, and also supply new and specific functionality for the *Circus* verification. The class `circus.TypeChecker` groups and instantiates the visitors of the *Circus* type checker, allowing communication among them; instantiates the *Circus* type environment; and stores the list of errors detected in the verification. The class `circus.Checker` is abstract; it defines the general behavior of the other visitors. It joins the common methods of the visitors and it has a reference to the `TypeChecker` class. With this reference, each variation of the `Checker` can communicate with each other.

For each Z visitor, there is a *Circus* visitor that implements visitation methods only for the additional constructors of *Circus* and for the Z constructors that need a special verification in *Circus*. Each *Circus* visitor extends the class `circus.Checker` and has a reference to the corresponding Z visitor. This reference allows the delegation of verification tasks to the Z type checker, when necessary.

The general behavior of the type checker is similar to that implied by the type rules. Only some details have been implemented in a different way, with the goal to follow the pattern of implementation of the CZT type checker. For example, in the implementation we do not extract all the process names of a specification nor all the action names of a process before initiating the verification, as the rules suggest. We let the verification occur sequentially and, in the case of some reference problem involving an action or a process name, the reference is checked at the end of the verification. These references are treated by the *visitor* `circus.PostChecker`. At this point, since all the specification has already been analyzed, certainly all the process or action declarations are stored in the environment. If there is still some reference to a process or action not declared, the type checker can now conclude that the specification has type errors.

The use of the visitor design pattern and of the CZT framework was very important for the integration with other *Circus* tools. Besides the type checker, several *Circus* tools have also been implemented using this framework: the model checker [7] and *JCircus* [6] are some examples. This design decision makes possible the use of the type checker as an isolated component. In this way, maintenance or evolution of the type checker does not require modifications in the other tools.

5 Validation

For the validation of the *Circus* type checker we chose two strategies. The first one was the elaboration of tests, to examine the behavior of the type checker through its isolated execution. We carried out tests using small specifications, to verify specific behaviors of the tool, and large specifications, like the *SmartCard* specification presented in [26] and the fire control system presented in [16]. To execute the tests, we used the JUnit tool to carry out unit and regression testing. It provides a test runner with a graphical interface that helps to visualize which test were successful and which were not. The use of JUnit for the execution of regression testing was important to make sure that changes in the type checker, such as bugfixes, did not have an impact on the tests for which the tool was already working correctly.

Another validation strategy was the integration of the type checker with other *Circus* tools namely, *JCircus* [6] and CircusRefine. These integrations have also served as acceptance tests for the type checker.

5.1 Integration with JCircus

JCircus is a tool that automates the rules that translate programs written in *Circus* to Java [6].

The type annotation performed by the type checker is of extreme importance for the *Circus* tools that need type information during their execution; *JCircus* is one of them. To find the type of a channel, it is enough to access the annotation of the AST node that represents the channel. If the type checker only had as functionality the verification of type consistency of a specification, the *Circus* tools would have to implement annotation facilities if they needed type information, and this, basically, requires a new verification.

The integration with *JCircus* allowed the identification of some problems in the type checker implementation. We detected the absence of some type annotations for some syntactic structures of *Circus*. Also, we identified that some error messages generated by the type checker were not clear.

5.2 Integration with CircusRefine

As we know, *Circus* has an associated refinement calculus that defines refinement strategies for processes and their actions, and provides refinement techniques for concurrent and distributed programs. The basic notion of refinement in *Circus* is the refinement of actions. The refinement of processes is established in terms of refinement of the main actions of these processes. In [2], we can find some of the refinement laws of processes and actions.

We have developed a *Circus* refinement tool to integrate with the type checker; it supports the application of the refinement laws of actions, which makes it possible to illustrate the integration and utility of the *Circus* type checker. Its architecture has been especially designed to allow future extension.

CircusRefine was developed based on another refinement tool, which supports Morgan's refinement calculus [14]: the Refine [18,28] tool. Graphically, these tools are similar. However, the processing logic is very different, because the tools use distinct parsers and were developed for distinct languages. CircusRefine is based on the Windows pattern, with screens, menus and buttons to access the various services. The tool recognizes (and receives as input) *Circus* specifications written in LaTeX, refinement laws of actions described using a template, and displays the steps of the development through its graphical interface.

CircusRefine carries out a type verification on each refinement law when it is loaded. The goal of this verification is to guarantee consistency of the specification after the application of laws. Consequently, if a law application does not request the input of arguments by the users, it is not necessary the type verification of all specification after the refinement.

The type checker of refinement laws is based on the *Circus* type checker and, as consequence, on the CZT type checker. The purpose of the development of this law type checker is to show that the *Circus* type checker also can be extended to develop other functionalities or tools.

In the integration with the CircusRefine tool, we identified the necessity of partial type verification that, in fact, consists of an improvement to be implemented. Currently, when a law is applied, and it requires the input of arguments, CircusRefine requests a complete type verification of the specification. However, only a specific part of the specification is updated by a law application, and so, it is not necessary a complete type verification. Currently, the type checker needs to analyze all the specification to be able to fill the environment with type information and create the verification contexts. To provide a partial verification, this type environment must be filled with the global information of types, and the type checker must identify the fragment contexts to carry out a correct and efficient verification.

6 Conclusion

In this paper, we presented the formal definition of the *Circus* type system. We defined the set of well formed type environments and presented some of the type rules. The defined rules are linked to the definition of the type system of Z standard. In total, we specified 118 *Circus* type rules. The complete list can be found in [26].

Another important contribution of our work was the implementation of a *Circus* type checker. The development of the type checker was based on the CZT framework, which makes possible the independent and organized construction of Z tools. We presented the extension of the CZT type checker for the implementation of the *Circus* type checker.

The validation of the *Circus* type system and its type checker was carried out through tests and integrations with other developed tools of the language. We followed the integration process of the type checker with the *JCircus* tool, and we developed an initial version of the refinement tool of *Circus*, called CircusRefine, which also uses the type checker as a component. With the integrations we could prove and improve the benefits and the functionalities offered by the type checker.

With the integrations, we detected that the type annotation is a resource of great utility for the other *Circus* tools and it prevents additional processing to be realized by them. Positively, we also confirmed that the integration with the type checker is very simple. The integration with the *JCircus* tool indicated some imperfections of the type checker implementation, such as absence of some type annotation and imprecise error messages. The integration with the CircusRefine tool identified the necessity of partial type verification, which we intend to do as future work. We also intend to develop the complete version of the *Circus* refinement tool. The current version of CircusRefine allows only the refinement of actions. The implementation of the type checker and CircusRefine can be found in [27].

The definition of the *Circus* type system and the implementation of its type checker are an important contribution to the evolution of *Circus*, clarifying essential points of its definition as a strongly typed language that is compatible with Z and CSP. Additionally, we are also contributing to the development of other tools for *Circus*. We hope that our work can serve as base for the definition and implementation of the type systems of the *Circus* extensions.

Concerning related work, as already mentioned, we are not aware of any complete type system for CSP based on formal rules. Regarding Z, we could substantially reuse its type system to formalise and implement an original type system for *Circus*. We are also unaware of formally developed type systems for any language that integrates process algebras and model based specifications.

References

[1] L. Cardelli. *Type Systems.* The Handbook of Computer Science and Engineering, 2nd Edition, Chapter 97. CRC Press, 2004.

[2] 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.

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

[4] C. Fischer. *Combining CSP and Z.* Technical Report, University of Oldenburg, Germany.

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

[6] A. F. Freitas.*From Circus to Java: Implementation and Verification of a Translation Strategy.* Master's thesis, Department of Computer Science, The University of York, 2005.

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

[8] C. A. R. Hoare and J. He. *Unifying Theories of Programming.* Prentice-Hall, 1998.

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

[10] J. Hoenicke and E.-R. Olderog. Combining Specification Techniques for Processes, Data and Time. In M. J. Butler, L. Petre, and K. Sere, editors, *IFM 2002: Integrated Formal Methods, Third International Conference*, volume 2335 of LNCS, pages 245–266. Springer, May 2002.

[11] W. Li, X. Zhou and S. Li. The Typing of Communicating Sequential Processes. *TOOLS '99: Proceedings of the 31st International Conference on Technology of Object-Oriented Language and Systems*, pages 61–66. IEEE Computer Society, 1999.

[12] B. P. Mahony and J. S. Dong. Deep Semantic Links of TCSP and Object-Z: TCOZ Approach. *Formal Aspects of Computing*, 13(2):142–160, 2002.

[13] T. Miller, L. Freitas, P. Malik, M. Utting. CZT Support for Z Extensions. Proc. *5th International Conference on Integrated Formal Methods (IFM 2005)*, Eindhoven, The Netherlands, pp227–245, Springer.

[14] C. C. Morgan. *Programming from Specifications.* Prentice-Hall, 2nd edition, 1994.

[15] P. Malik and M. Utting, CZT: A Framework for Z Tools. In Treherne, H., King, S., Henson, M., and Schneider, S., editors, ZB2005: Formal Specification and Development in Z and B, *4th International Conference of B and Z Users*, Guildford, UK, April 2005., pp 65–84, Springer-Verlag, LNCS 3455, 2002.

[16] M. V. M. Oliveira, A. L. C. Cavalcanti and J. C. P. Woodcock. *Refining Industrial Scale Systems in Circus.* In I. East, J. Martin, P. Welch, D. Duce and M.Green, editors, Communicating Process Architectures, pages 281–309, 2004.

[17] M. V. M. Oliveira. *Formal Derivation of State-Rich Reactive Programs using Circus.* PhD thesis, Department of Computer Science – University of York, UK, 2005. YCST–2006–02.

[18] M. V.M. Oliveira, M. A. Xavier, and A. L. C. Cavalcanti. Refine and Gabriel: Support for Refinement and Tactics. In J. R. Cuellar and Z. Liu, editors, *2nd IEEE International Conference on Software Engineering and Formal Methods*, pages 310 – 319. IEEE Computer Society Press, September 2004.

[19] G. Smith. The Object-Z Specification Language. Advances in Formal Methods. *Kluwer Academic Publishers*, 2000.

[20] G. Smith. An integration of Real-Time Object-Z and CSP for Specifying Concurrent Real-Time Systems. In M. Butler, L. Petre, and K. Sere, editors, *IFM 2002*, page 267–285. Springer-Verlag, 2002.

[21] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, 2nd edition, 1992.

[22] K. Taguchi and K. Araki. The state-based CCS semantics for concurrent Z specification. In M. Hinchey and Shaoying Liu, editors, *International Conference on Formal Engineering Methods*, pages 283–292. IEEE, 1997.

[23] J. C. P. Woodcock and A. L. C. Cavalcanti. *A Concurrent Language for Refinement.* In 5th Irish Workshop on Formal Methods, 2001.

[24] J. C. P. Woodcock and A. L. C. Cavalcanti. *The Semantics of Circus. In ZB 2002: Formal Specification and Development in Z and B*, volume 2272 of Lecture Notes in Computer Science, pages 184–203. Springer-Verlag, 2002.

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

[26] M. Xavier. *Definição e Implementação do Sistema de Tipos da Linguagem Circus.* Master's thesis, Centro de Informática, Universidade Federal de Pernambuco, 2006. http://www.cin.ufpe.br/~max/MSc/dissertation.pdf.

[27] M. A. Xavier. *Formal Definition and Implementation of Circus Type System*, 2006. http://www.cs.york.ac.uk/circus/type-checker/xavier-msc/.

[28] M. A. Xavier and A. L. C. Cavalcanti. Mechanised Refinement of Procedures. In *8th Brazilian Symposium on Formal Methods 2005*, pages 47–62. Porto Alegre, Brazil, 2005.

[29] ISO/IEC, Information Technology – Z Formal Specification Notation – Syntax, Type System and Semantics, ISO 13568:2002 International Standard, 2002.