A Formal Semantics for the Language Genesis: An Application of the Z Notation and Plotkin's Structured Operational Semantics Style

Jorge Henrique Cabral Fernandes

Universidade Federal do Rio Grande do Norte
Departamento de Informática e Matemática Aplicada
Centro de Ciências Exatas e da Terra - Campus Universitário - Natal - RN - BRASIL
CEP 59.072-970
`jorge@dimap.ufrn.br`

Abstract. This paper presents a formal semantics for the language Genesis, using the Z notation. Genesis was developed after the Corpus model, which is based on elements of the Actors model and event-based architectures. Genesis explores the quest for open systems by enabling the hierarchical design and construction of large to small scale open computing systems. Genesis caters for the openness of the systems it builds by enabling the exchange of asynchronous communications in a loose federation of distributed autonomous computing components. Beyond informally presenting simple aspects of the language, this paper describes the overview organization of a large and complete example of how to use the Z notation. Additionally the paper shows how to employ Plotkin's Structured Operational Semantics (SOS) style for the description of a concurrent and parallel programming language.

1 Introduction

This paper aims to show the overview organization of a large and mostly complete example of how one can use the Z notation for specification of syntactical and semantical aspects of a parallel and concurrent programming language. It also shows how one can use the Plotkin's SOS style for specification of concurrent features of a parallel computing system.

We hope that by explaining the overall organization of the work, by pointing to sections of the complete and detailed specification and finally by making widely available the LaTeX sources of the specification, we may enhance the rate of interest and adoption of the Z notation for specification of language semantics and of the SOS style for specification of concurrent transition systems.

Furthermore, this work introduces the Genesis language, a language aimed for the description of open systems obeying to the rules of the Corpus model, which may be of interest to readers acquainted with families of Actor languages.

The paper is organized as follows. Section 2 presents a quick overview of the Corpus computing model. Section 3 briefly presents the language Genesis. Section 4 presents an overview of the semantics specification. It departs from an abstract syntax for the language and gradually introduces an overall model for intra component computation, ending with a description of the use of the SOS style. The specification ends with the presentation of the *meaning* function. Finally, Section 5 discusses our results.

2 Corpus

Genesis is a programming language conceived for the construction of systems obeying the properties of the Corpus model. Corpus is a novel computing model for distributed, concurrent, dynamical and event-driven computing, aiming to satisfy a set of requirements for open system models such as emergence, edge of chaos, imprecise system goals and imperfect knowledge. Corpus is organized around the ideas of the Actors model[1] and principles found in event-based approaches.

The core concepts and definitions of the Corpus model are: (1) **corpus** - an autonomous, uniquely identified, computational entity that presents a computational state (type name, local store, running threads, thread templates, mailboxes and store of known type definitions) plus a communication state (communications enqueued); (2) **corpora** - set of corpus that are related through an acquaintance relationship. A corpus acquaints another corpus if the former has a reference to the external mailbox of the latter; (3) **external environment** of a corpus - set of corpus that acquaints this corpus. (4) **internal environment** of a corpus - set of components that the corpus acquaints. (5) **messages and events** - kinds of communications that flow through the corpora. Messages flow from one corpus to the external mailbox of one's acquaintance. Events flow from the acquainted to all the corpus that compose its external environment; (6) **no self-reference** - a corpus does not hold an explicit reference to itself nor to the components that make up its external environment; (7) **dynamism** - references to a corpus may be sent in communications. Thus the configuration of a corpora may change over time; (8) **asynchrony** - the production and consumption of communications is temporally decoupled; (9) **type definition** - description of the computational and communication properties to be satisfied by a corpus; (10) **corpus creation** - each corpus is created by other (creator) corpus, based on a known type definition; (11) **receptionists** - foreign computing components may be seamlessly integrated into a corpora, provided that it externally obeys the properties of the model.

Figure 1 represents a corpora composed by two corpus, A and B, analyzed from the point of view of A. B is an acquaintance of A, and, on the converse side, A makes up the external environment of B. The external environment of A is unknown. A may raise events and receive messages from its external environment. Given that A holds a reference to B, it may send messages to B and handle events produced by B.

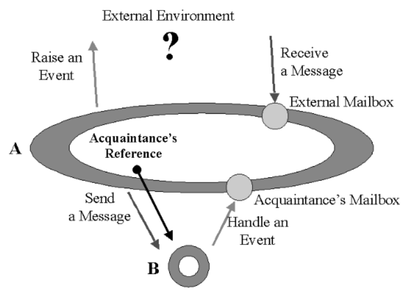

Fig. 1. A Corpus and its Communicational Capabilities.

General properties of open computing systems derived from the Corpus model, such as reliability through use of mediators and dynamic architectures, are further explored in [3]. The rest of the paper focuses on Genesis and its semantics.

3 Genesis

A Genesis program describes a way to build an (open) computing system based on Corpus (a corpora). Each Genesis program is composed by a (possibly open and dynamical) set of type definitions. Each type definition may be seen as the specification of the structure and behavior that characterize that type of component.

The concrete syntax of a type definition in Genesis is as follows:

$$\begin{aligned} \mathit{def} &::= \texttt{DEF}\ \ \mathit{defName}\ =\ \{\ \ \mathit{varsDecl}\ \ ;\ \ \mathit{templatesDecl}\ \ \} \\ \mathit{varsDecl} &::= \texttt{VARS}\ \ \mathit{varName}_1,\ \ \mathit{varName}_1,\ \ ...,\ \ \mathit{varName}_n \\ \mathit{templatesDecl} &::= \texttt{TEMPLATES}\ \ \mathit{template}_1;\ \ \mathit{template}_2;\ \ ...;\ \ \mathit{template}_n; \end{aligned}$$

A definition (def) is composed by a name ($\mathit{defName}$), a sequence of (names of) variables that comprises the local store of the component, and a sequence of thread templates.

Thread templates describe how to create threads, according to communications in which the component engages. A template is an anonymous statement that may nest blocks formed by a sequence of statements. The general syntax of a template is as follows:

$$\begin{aligned} \mathit{template} &::= \mathit{stmt}\ \ [\Rightarrow \mathit{stmtBlock}] \\ \mathit{stmtBlock} &::= \mathit{stmt}_1;\ ...\ ;\ \mathit{stmt}_n \end{aligned}$$

The statements that make up templates are grouped under five kinds:

- let bindings;
  $\mathit{let} ::= \texttt{LET}\ \ \mathit{name}\ =\ \mathit{expr}\ \ [\Rightarrow \mathit{stmtBlock}]$

Computes the value of named private expressions which are added to the scope of the block. Once computed, the (name, value) binding is unchangeable.

- waiting for an incoming communication;

  guard ::= [`LET` *nameList* .] `ON` *acqName* . *evtName* (*pattern*) [$\Rightarrow$ *stmtBlock*]
  | [`LET` *nameList* .] `IN` *msgName* (*pattern*) [$\Rightarrow$ *stmtBlock*]

  An `ON` statement defines the event pattern that matches the event that this component intends to handle from an acquaintance. An `IN` statement defines the message pattern that matches with the message that this component intends to receive from the external environment. There is no identification of the sender, for the components that comprise the corpus's external environment cannot be discriminated. Names in *nameList* are bound to values carried into the communications received.
- producing an outgoing communication;

  outgoingComm ::= `SEND` *acqName* . *msgName* (*exprList*) [$\Rightarrow$ *stmtBlock*]
  | `OUT` *msgName* (*exprList*) [$\Rightarrow$ *stmtBlock*]

  A `SEND` statement creates a message that is sent to an acquaintance. An `OUT` statement creates an event that is multicasted to all components that form the corpus' external environment.
- store assignment;

  assignment ::= *varName* := *expr* [$\Rightarrow$ *stmtBlock*]

  The value of the expression is computed and stored into the variable.
- control;

  cond ::= `IF` $boolExpr_1 \Rightarrow stmtBlock_1$
  [`ELSIF` $boolExpr \Rightarrow stmtBlock_i$]*
  [`ELSE` $\Rightarrow stmtBlock_n$]

  This statement has the usual meaning adopted in most programming languages. Templates cannot begin with an `IF` statement.

Obeying to the rules for Actor languages, all computation in Genesis is driven by the communications that a component receives. Pushing this directive to its theoretical limits, Genesis does not has any loop or goto statement, so that infinite computation in Genesis may arise only if there is an infinite amount of communications flowing into the system.

For the sake of an example, the type definition `Nat` describes a corpus that contains a store variable named `value` and defines three thread templates.

```
DEF Nat = {
  VARS value;
  TEMPLATES
    value := 0;
    IN succ() => value := value + 1 => OUT setValue(value);
    LET newV. IN setValue(newV) =>
      value := newV =>
        OUT setValue(value);
}
```

At initialization, a (corpus of type) `Nat` assigns 0 to the variable `value`. Each time it receives a message `succ`, a thread is created and produces the following behavior: (1) `value`'s value is incremented by one and; (2) an event `setValue`, holding the new value stored in `value`, is raised to the external environment. Each time it receives a message `setValue`, the value hold by the message is assigned to the local name `newV` and a thread is created. The effect of the thread is as below: (1) `newV`'s value is assigned to `value` and; (2) an event `setValue`, holding the new value is raised.

While the `Nat` example is meant to give a clue for the informal concrete syntax and semantics of the language, the combination among the statements above, whether in the same thread template, whether in different templates, expresses linear or inherently parallel programming constructions that support a large diversity of behaviors. Chapters 5 and 6 and Appendix 2 of [3] presents several examples of running programs, some meant to explore possible language constructions, and others that, integrated with externally developed Java components (receptionists), lead to the construction of interesting and useful systems such as Petri Net simulators, chatting systems, GUI event handling, digital systems simulators, etc. Actual use of Genesis relies on a toolkit formed by a language compiler, virtual machine and distributed execution monitor.

The remainder of this paper concentrates on the organization of the formal semantics.

4 Organization of the Specification

The syntax, messages, threads, families of threads, components, configurations, transitions, etc, which make up the specification, are described using Z[9, 8, 7]. Several aspects of the Z style of this specification were based on the work of Marc Benveniste[2]. The specification has been fully syntax and type checked using the Z/EVES tool[6].

The semantics was built after the construction of toolkit and thus it is heavily biased towards its actual implementation. It describes a formal and detailed interpreter (an abstract one-step machine) for Genesis systems and a way to map a given Genesis script into structures of this interpreter. Therefore, the semantics here presented is a Concrete Computation Semantics, in the sense advocated by [4].

Genesis components present an inherently concurrent nature. Thus, each corpus has its own autonomous abstract machine. Furthermore, the set of components (corpora) that make up a global configuration may possibly engage in state transitions by exchanging messages and events. This part of the semantics follows an operational style similar to that advocated by Gordon Plotkin[5].

The semantics is comprehensive and covers most aspects of the language, but some elements were left unspecified, such as literal guards and lexical inclusion. Additionally, most features of the Genesis toolkit are also left unspecified. These simplifications were done mostly due to space and time limitations. The reader is urged to read [3] for more details about these aspects.

4.1 Overall Organization of the Semantics

Figure 2 presents an overall structure of the semantics. It pictures the main parts of the specification, their subsections and dependencies. The full specification is composed by the following amount of Z Elements:

- 151 schemas;
- 28 axiomatic descriptions;
- 8 free types;
- 6 given sets, and;
- 4 abbreviations.

These elements are structured around five (5) parts which are further split into thirty four (34) subsections. When referring to the parts of the specification we will use the original names of the sections as presented in Appendix 1 of [3].

Part A.2 describes an abstract syntax for Genesis. This comprises the declaration of names, expressions, statements, templates, definitions and scripts. **Expressions** use **Names**, **Statements** compose expressions in well formed rules, **Templates** are special kinds of statements. A **Definition** declares a store plus the templates that may update this store, and a **Script** is a set of definitions.

Part A.3 basically describes structural model for a **Thread**. Which involves the description of models for:

- **Values** that are held inside threads;
- scope and reachability relations for **Access** to names in expressions and statements;
- free and bounded communication **Handlers**, not discussed in this paper due to space limitations;
- **Complex Stores** of a thread, not discussed in this paper due to space limitations.

Part A.4 describes structural and behavioral aspects of the computation that occurs inside a **Corpus**. The structural aspects comprise descriptions of models for:

- **Events and Messages**;
- **Store of a Component**, which may hold queues of pending messages;
- **Thread Families** that manage the creation of threads;

The behavioral aspects of a component's computation involve the definition of rules for **Evaluation of Expressions** and **Execution of Statements**. These rules obey changes over the state of a component, and rely on **Framing Schemas** to help make more explicit these changes.

Part A.5 describes a semantics for **Reception of Incoming Communications**, which involves:

- rules for **Matching a Communication with a Handler**;
- rules for **Selection of the Thread** that is to be notified of the arrival of the communication. This also involves **Selecting the Family** that will manage the threads's creation and destruction.

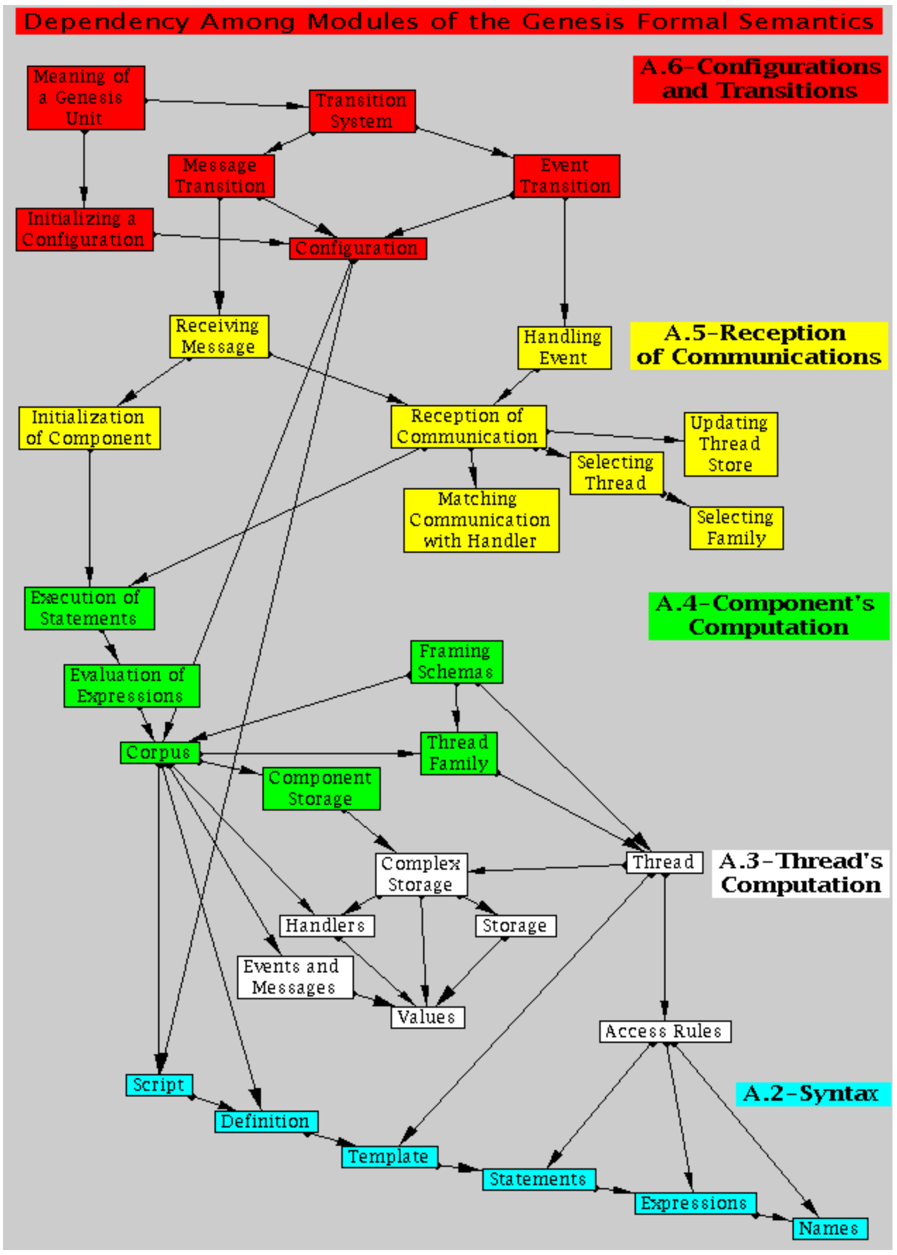

Fig. 2. Overall Structure of the Genesis Formal Semantics.

– rules for **Updating the Private Store of the selected Thread**;
– the execution of the statements which follow the communication handling statement in the thread.

Furthermore, two different kinds of communications may be received by a component, and this involves the description of rules for **Receiving a Message** and **Handling an Event**. The reception of a message may involve the **Initialization of the Component**, which may trigger the execution of statements.

Finally, Part A.6 provides a model for description of the inherently concurrent aspects of the Genesis language. This comprises the definition of a **Configuration** composed by a set of autonomous components, and rules for description of the transitions that may happen while these components exchange communications. A transition implies in the reception of a message by a target component or the handling of the event by a set of targets components.

The ending part of the semantics provides a **Meaning for the Execution of a Genesis Unit** of work. The meaning is defined by the **Initialization of a Configuration** and by the construction of a trace relation that describes the possible sequence of changes in the global configuration of the system, obeying to the rules of the Transition System.

4.2 Remarks on the Style of the Specification

Under the point of view of usage of Z styles, this semantics might be divided into five styles.

The First Style: Free Types for Description of the Syntax Used in Part A.2, it describes the syntactic elements of Genesis, which have been mostly done through the use of given types, abbreviations and free types, such as the sketched below.

$$\begin{aligned}&[\mathit{VariableName}, \mathit{PrivateName}, \mathit{EventName}, \mathit{MessageName}]\\&\mathit{Literal} == \operatorname{seq}\mathbb{N}\\&\mathit{DefName} == \mathit{Literal}\end{aligned}$$

Given types such as *VariableName*,*PrivateName*,*EventName* and *MessageName* introduce atomic syntactical elements. Abbreviations such as *Literal* and *DefName* provide more readability and abstractness to the specification.

$$\begin{aligned}\mathit{Expr} ::=\;& \mathit{null}\\ |\;& \mathit{litExpr}\langle\langle \mathit{Literal}\rangle\rangle\\ |\;& \mathit{equals}\langle\langle \mathit{Expr}\times\mathit{Expr}\rangle\rangle\\ |\;& \ldots\\ \mathit{Stmt} ::=\;& \mathit{if}\langle\langle \mathit{Expr}\times\operatorname{seq}\mathit{Stmt}\times\operatorname{seq}\mathit{Stmt}\rangle\rangle\\ |\;& \mathit{send}\langle\langle \mathit{Expr}\times\mathit{MessageName}\times\operatorname{seq}\mathit{Expr}\times\operatorname{seq}\mathit{Stmt}\rangle\rangle\\ |\;& \ldots\end{aligned}$$

Free types such as *Expr* and *Stmt* were used to map objects from a type into other type. For instance, the skeleton of *Expr*, shown above, declares the *null* object to be of type *Expr*. In the second line, *litExpr* is declared to be a function that maps a *Literal* object into an *Expr* object. Additionally, the function *equals* maps two objects of type *Expr* into a (more complex) object of type *Expr*. In the skeleton of *Stmt* the functions *if* and *send* are declared to map compositions of expressions (*Expr*), sequences of statements (seq *Stmt*) and other types into objects of type *Stmt*. Free types are thus useful for description of union and recursive types.

```
SENDStmt
  stmt : Stmt
  acqName : Expr
  msgName : MessageName
  exprList : seq Expr
  block : seq Stmt
  ----------------------------
  stmt = send(acqName, msgName, exprList, block)
  ∃ pName : PrivateName; vName : VariableName •
      acqName = privExpr(pName) ∨ acqName = varExpr(vName)
```

Complementing the use of free types, schemas such as *IFStmt* and *SENDStmt* are useful for extracting a record structure from free types, introducing variables that explicitly name all elements that make up a statement. For instance, the schema *SENDStmt* introduces an *Stmt* named *stmt* and declares the variables *acqName*, *msgName*, *exprList* and *block* as names for the fields that compose a *stmt*. Furthermore, the predicate part of *SENDStmt* imposes some constraints on a *send* statement, by declaring that the *acqName* expression is an special kind of expression, which access a *PrivateName* or *VariableName*.

The Second Style: Axiomatic Definitions for Generation of the Interpreter's State Present in Parts A.3 and most of A.4, it basically describes the static structural features of a component and its constituent aggregates. The definition of the structure of the interpreter is done through schemas such as *PrivateStore*, *Thread*, *ThreadFamily* and *Corpus*, which help to describe the interpreter's static properties and constraints.

This style relies heavily on the use of axiomatic definitions such as *makePrivateStore*, *makeThread*, *makeThreadFamily* and *makeCorpus* for describing the **process** of construction of the interpreter's state. These axiomatic definitions may be thought of as providing a constructive process for generation of the state of the interpreter, which maps syntactic features (*Template*, *TemplateCtx*, *Definition*) into interpreter features (*Thread*, *ThreadFamily* and *Corpus*). The skeletons of the schema *ThreadFamily* and of the axiomatic definition *makeThreadFamily* help to illustrate this style.

$$\begin{array}{|l}
\textit{ThreadFamily} \\
\hline
\textit{TemplateCtx} \\
\textit{id} : \mathbb{N} \\
\textit{threads} : \mathbb{N} \nrightarrow \textit{Thread} \\
\textit{nextThreadNum} : \mathbb{N} \\
\hline
... \\
\hline
\end{array}$$

$$\begin{array}{|l}
\textit{makeThreadFamily} : \textit{TemplateCtx} \times \mathbb{N} \nrightarrow \textit{ThreadFamily} \\
\hline
\forall\, \textit{ctx} : \textit{TemplateCtx};\ \textit{familyId} : \mathbb{N} \bullet \exists\, f : \textit{ThreadFamily} \bullet \\
\quad\quad \textit{makeThreadFamily}(\textit{ctx}, \textit{familyId}) = f \land \\
\quad\quad (\exists\, \textit{statement} : \textit{Stmt} \mid \textit{statement} = \textit{ctx}.\textit{statement} \bullet \\
\quad\quad\quad\quad ((\textit{GuardTemplate} \Rightarrow ...) \\
\quad\quad\quad\quad (\textit{InitTemplate} \Rightarrow ...)))
\end{array}$$

The Third Style: Schemas Define Rules for Evaluation of Individual Expressions and Statements Given the availability of a supporting structure for the interpreter, the remainder of Part A.4 describes rules for evaluation of expressions and statements, which are defined in several schemas, one for each kind of expression (*GetNullValue*, *GetLitValue*, *GetEqualsValue*, etc) and statement (*IFStmtExecution*, *SENDStmtExecution*, etc) of the language.

For instance, the schema *GetLitValue* describes how to evaluate an expression of the form $\textit{litExpr}(\textit{literal})$. It receives an input expression (denoted by $\textit{expr}?$) and outputs a value (denoted by $v!$).

$$\begin{array}{|l}
\textit{GetLitValue} \\
\hline
\textit{expr}? : \textit{Expr} \\
v! : \textit{Value} \\
\hline
\exists\, \textit{literal} : \textit{Literal} \mid \textit{expr}? = \textit{litExpr}(\textit{literal}) \bullet \\
\quad\quad v! = \textit{litValue}(\textit{literal}) \\
\hline
\end{array}$$

The execution of statements relies on the evaluation of its constituents expressions and nested sequences of statements, and describes changes to be applied to the state of an specific thread that belongs to an specific family inside an autonomous component. See for instance the skeleton of the schema *IFStmtExecution*.

$$\begin{array}{|l}
\textit{IFStmtExecution} \\
\hline
\textit{statement} : \textit{Stmt} \\
t, t' : \textit{Thread} \\
f, f' : \textit{ThreadFamily} \\
c, c' : \textit{Corpus} \\
\textit{corp}! : \mathbb{P}\ \textit{Corpus} \\
\hline
\exists\, \textit{IFStmt} \mid \textit{statement} = \textit{stmt} \bullet ... \\
\hline
\end{array}$$

The statement to be evaluated is denoted by the variable *statement*, and (as defined in the predicate part of the schema) must to be an *IFStmt*. The previous state of the thread, family and component is denoted by variables t, f and c, respectively. The execution of *statement* produces changes over the state of the thread, family and component, which are denoted by variables t', f' and c', respectively. Furthermore, new components produced during the evaluation of *statement* are denoted by the set of *Corpus* named $corp!$.

Two axiomatic definitions named *getValue* and *allStatementsExecution* act as functions that centralize the evaluation of all expressions and statements. As it is shown below, their signatures are very similar, for both may produce changes over the state of a component.

$$\begin{array}{|l}
getValue : (Expr \times Thread \times ThreadFamily \times Corpus) \rightarrow\!\!\!\!\!+ \\
\qquad\qquad\qquad (Value \times Thread \times ThreadFamily \times Corpus \times \mathbb{P}\,Corpus) \\
\hline
\forall\, expr? : Expr;\ t : Thread;\ f : ThreadFamily;\ c : Corpus \bullet \\
\ \exists\, v! : Value;\ t' : Thread;\ f' : ThreadFamily; \\
\qquad\qquad\qquad c' : Corpus;\ corp! : \mathbb{P}\,Corpus \bullet \\
\qquad ((v!, t', f', c', corp!) = getValue(expr?, t, f, c) \wedge \\
\qquad (GetNullValue \wedge ThreadUnchanged \wedge CorporaUnchanged) \vee \\
\qquad ...)
\end{array}$$

$$\begin{array}{|l}
allStatementsExecution : (Thread \times ThreadFamily \times Corpus) \rightarrow\!\!\!\!\!+ \\
\qquad\qquad\qquad\qquad (Thread \times ThreadFamily \times Corpus \times \mathbb{P}\,Corpus) \\
\hline
\forall\, t : Thread;\ f : ThreadFamily;\ c : Corpus \bullet \\
\qquad \exists\, t' : Thread;\ f' : ThreadFamily;\ c' : Corpus;\ corp! : \mathbb{P}\,Corpus \bullet \\
\qquad\qquad (allStatementsExecution(t, f, c) = (t', f', c', corp!) \wedge \\
\qquad\qquad (\exists\, t1 : Thread;\ f1 : ThreadFamily;\ c1 : Corpus; \\
\qquad\qquad\qquad statement : Stmt;\ corp1 : \mathbb{P}\,Corpus \mid CurrentStatement \bullet \\
\qquad\qquad\qquad ((IFStmtExecution[t1/t', f1/f', c1/c', corp1/corp!] \vee \\
\qquad\qquad\qquad ...))))
\end{array}$$

The Fourth Style: Schemas Describing the Reception of Communications Part A.5 relies on several schemas to define the effect that the reception of communications provokes on a component. The three core schemas of this section are *HandleComm*, *ComponentReceiveMessage* and *ComponentHandleEvent*, whose skeletons are shown below.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ HandleComm\ }} \\
comm : Comm \\
h : Handler \\
c, c' : Corpus \\
corp! : \mathbb{P}\,Corpus \\
pHandler! : \mathbb{P}\,Handler \\
\hline
... \\
\hline
\end{array}$$

```
— ComponentReceiveMessage ——————————————
| comm : Comm
| c, c′ : Corpus
| corp! : ℙ Corpus
|————————————
| ...
|———————————————————————————————————————
```

$$\begin{array}{|l}\hline \textit{ComponentReceiveMessage} \\ comm : Comm \\ c, c' : Corpus \\ corp! : \mathbb{P}\ Corpus \\ \hline \ldots \\ \hline \end{array}$$

$$\begin{array}{|l}\hline \textit{ComponentHandleEvent} \\ comm : Comm \\ source : CorpusRef \\ c, c' : Corpus \\ corp! : \mathbb{P}\ Corpus \\ \hline \ldots \\ \hline \end{array}$$

HandleComm is used by *ComponentReceiveMessage* and *ComponentHandle-Event*. It takes a communication ($comm$), a handler (h) and a corpus (c) and produces a changed corpus (c'), a set of newly produced corpus ($corp!$) and a set of handlers ($pHandler!$, of cardinality 1) that is empty case h is to be discarded after first use. *ComponentReceiveMessage* takes a communication ($comm$) and a corpus (c) and produces a changed corpus (c') and a set of newly created corpus ($corp!$). *ComponentHandleEvent* is similar to *ComponentReceiveMessage* but differs on the knowledge of the sender of the communication, denoted by $source$.

Only one axiomatic description ($startCorpus$) is employed in this part, just for the purpose of initialization of the component. The signature of $startCorpus$, shown below, indicates that the initialization of a component (c) results in a modified component (c') and a possibly null set of new components ($corp'$), which are created during the initialization of c.

$$\begin{array}{|l} startCorpus : Corpus \rightarrowtail\!\!\!\!\!+ (Corpus \times \mathbb{P}\ Corpus) \\ \hline \forall\, c : Corpus \bullet \exists\, c' : Corpus;\ corp' : \mathbb{P}\ Corpus \bullet \\ \qquad (startCorpus(c) = (c', corp') \wedge \\ \qquad \ldots) \end{array}$$

The Fifth Style: Configuration and Transitions Employed in the Part A.6, it is similar to that employed in the second part, but it is more simpler, for just two new structures are defined: the structure of a configuration (schema *Config* sketched below); and the structure of a Genesis unit of work (schema *Unit*), this last denoting a job possibly submitted by a user to a genesis virtual machine.

$$\begin{array}{|l}\hline \textit{Config} \\ corpora : CorpusRef \rightarrowtail\!\!\!\!\!+ Corpus \\ Script \\ \hline \ldots \\ \hline \end{array}$$

```
Unit
  $script? : \mathbb{F}\ Definition$
  $seedName? : DefName$
  ---
  ...
```

The remaining schemas that comprise most of this part cope with the rules for state changes to be applied to a configuration when two or more of its constituent components (that belong to the range of the function *corpora*) exchange their communications. The schemas *MessageTransition* and *EventTransition* (not shown here) describe this.

```
MessageTransition
  $cfg, cfg' : Config$
```

MessageTransition (and also *EventTransition*) take a configuration in a previous state (*cfg*) and map it to a changed configuration (*cfg′*) that represents the state of the system after the exchange of communications.

```
  $transitionSystem : \mathbb{P}(Config \leftrightarrow Config)$
  ---
  ...
```

The axiomatic definition *transitionSystem* defines a relation that may occur between all these possible configuration transitions.

$Trace == \mathbb{N} \mathrel{+\!\!\!\!\twoheadrightarrow} Config$

```
  $meaning : Unit \rightarrow Trace$
  ---
  $\forall u : Unit \bullet \exists t : Trace \bullet$
      $(meaning(u) = t \land$
      $(\exists initialCfg : Config \bullet$
          $t(0) = initialCfg \land$
          $StartSystem[initialCfg/cfg']) \land$
      $(\forall i : \mathbb{N}_1 \mid i \in \mathrm{dom}\, t \bullet$
      $(t(i-1), t(i)) \in transitionSystem) \land$
      $((\exists n : \mathbb{N}_1 \bullet n = \#(\mathrm{dom}\, t)) \Rightarrow$
          $\neg (\exists cfg : Config \bullet (t(\#(\mathrm{dom}\, t)), cfg) \in transitionSystem)))$
```

The definition *meaning* gives a meaning to a Genesis unit of work. It denotes a mathematical structure that represents the possible trace of changes in configurations that are possibly obtained during the execution of a Genesis unit. The first part of the predicate of the definition describes that the initial configuration of a trace is that obtained through the application of the constraints defined by *StartSystem*. The second part of the predicate constrains all consecutive pairs of configurations to belong to the relation *transitionSystem* described above. The last part of the predicate indicates that any finite trace of configuration changes must lead to a state where there are no further transitions possible. Informally this "pushes" the Genesis system towards the transition to a new possible state.

The Table 1 below provides a summary of key Z elements of the semantics.

5 Conclusions

This paper begins by briefly presenting informal aspects of the language Genesis. Nevertheless, the core of the paper holds its focus on the organizational structure adopted for description of a formal semantics for Genesis using Z, employing the technique of gradual introduction of details that is very akin to Z. The presentation of: (1) the overview picture (Figure 2) at the beginning of the formal semantics, and (2) the Table 1 with the summary of the language features (plus reference to the places where these elements are formally described) may help the reader into the comprehension of how to build a semantics for a concurrent language without delving into the initial details of the specification presented in [3].

The motivation for the construction of this paper arose from a real problem experienced by the author. After the design and implementation of the first version of the Genesis language and its execution environment, the author did not had available the LaTeX sources of a complete example of a concurrent language description (concrete and abstract syntax, formal semantics plus examples). Given this lack of resources, the author spent around six months adapting existing SOS specifications such as those described in [2] and [5] for the description of Genesis. Given that the author had previous experiences with Z, other interesting semantics works would require a significant amount of resources to translate to Z.

We hope that by making this paper and the detailed description freely available at the web we could contribute to the dissemination of this important and useful technique for the community of Z users.

There is also an important conceptual aspect that was not tackled in the semantics: the quest for openness in a Genesis system. Our *meaning* function provides an interpretation for a Genesis system by mapping a given *Unit* into a trace that contains the possible sequence of configurations reached during the execution of the resulting system. The closeness of the formal semantics relies in the fact that the *Unit* provided for the meaning function (beginning of the execution) denotes a **preexisting** set of definitions from which are further deduced all possible states to be reached by the system. This a priori knowledge about the possible space of states of a system conflicts with the notions of emergence, edge of chaos, imprecise system goals and imperfect knowledge that are pertinent to the concepts of open systems. We could provide a temporary patch to the problem by defining a reconfiguration *Unit*, a unit of higher scope than the original, which could join the meanings of two other sub-units producing a new modified meaning. This would only shift the problem to a (still closed system of) higher scope. Perhaps other theoretical frameworks are needed to properly

Feature	Z Element
Names for variables, private names events, messages and definitions	*VariableName*, *PrivateName* *EventName*, *MessageName* *CommName*, *DefName*
Abstract syntax for expressions	*Expr*
Abstract syntax and well-formedness rules for statements	*Stmt*, *IFStmt*, *SENDStmt*, *OUTStmt* *ASSIGNStmt*, *LETStmt*, *ONStmt* *ONFreeStmt*, *ONBoundedStmt*, *INStmt* *INFreeStmt*, *INBoundedStmt*
Genesis values	*Value*
Store Model for Threads	*PrivEntry*, *PrivateStore* *PartialVarEntry*, *VarStore*
Creation of Threads	*makeThread*, *makeThreadFamily* *CreateNextThreadInFamily*
Creation of ThreadFamily	*makeThreadFamily*, *makeComputation*
Events and Messages	*Event*, *Message*
Evaluation of Expressions	*GetVarValue*, *GetNullValue* *GetPrivValue*, *GetLitValue* *GetNewLiteralValue*, *GetEqualsValue* *GetPlusValue*, *GetMinusValue* *GetTimesValue*, *GetGreaterValue* and *getValue*
Assignment	*ASSIGNStmtExecution* *UpdateStoreInCorpus* *FlushPendingMessages*
Sending a Message	*SENDStmtExecution* *BlockPendingMessage*
Raising an Event	*OUTStmtExecution*
Creating a Component	*makeCorpus* *GetNewLiteralValue*
Installing Event Handlers	*ONStmtExecution* *InsertBoundedEventHandler* *makeVarFreeEventHandlers*
Installing Message Handlers	*INStmtExecution* *InsertBoundedMessageHandler* *makeFreeMessageHandlers*
Handling Events and Messages	*HandleComm* *ComponentReceiveMessage* *ComponentHandleEvent*
Exchanging Message	*MessageTransition*
Exchanging Event	*MulticastEventTransition* *EventTransition*
Starting Virtual Machine	*StartSystem*
Execution Traces	*meaning*

Table 1. Key Z Elements of the Genesis Semantics.

express this openness, which, however, does not reduces the importance of the following mechanical specification, fundamental for the precise comprehension of the language Genesis.

As an final remark it is important to state that during the process of building the specification, several problems and issues that existed in the initial semantics of Genesis (as defined by the implementation which was built before the formal description) were removed or clarified, which was invaluable for the enhancement of the consistence and completeness of Genesis.

References

1. Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.
2. M. Benveniste. Operational semantics of a distributed object-oriented language and its Z formal specification. Technical Report 1230, INRIA, 1990.
3. Jorge H. C. Fernandes. *Corpus/Genesis: an Approach for Construction of Distributed, Concurrent, Dynamical and Event-Driven Open Systems.* PhD thesis, Informatics Centre - Federal University of Pernambuco, 2000. Available: site DIMAp - UFRN. URL: http://www.dimap.ufrb.br/~jorge. Last accessed in Jul 2002.
4. Matthew Hennessy. *The Semantics of Programming Languages: An Elementary Introduction using Structural Operational Semantics.* John Wiley and Sons, 1990. Revised and condensed version.
5. Gordon Plotkin. A structural approach to operational semantics. research report daimi fn-19. Technical report, Department of Computer Science, Aarhus University, Aarhus, Denmark, 1981.
6. Mark Saaltink. The Z/EVES user's guide. tr-97-5493-06. Technical report, ORA Canada, 267 Richmond Road, Suite 100 - Ottawa, Canada, Sept 1997. Available at <http://www.ora.on.ca/~teste>.
7. J. M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics*, volume 3 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1988.
8. J. M. Spivey. *The Z Notation: A Reference Manual.* C. A. R. Hoare Series Editor. Prentice Hall, 1989.
9. Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof.* Prentice-Hall, London, UK, 1996.