



Denotational Semantics for Cloud# Language

By

Yehia Moustafa Abd AL-Rahman

Supervisor

Dr. Mourad Maouche

**This Thesis was submitted in Partial Fulfillment of the Requirements for the
Master's Degree in Computer Science.**

Deanship of Academic Research and Graduates Studies

Philadelphia University

January, 2013

جامعة فيلادلفيا

نموذج التفويض

أنا يحيى مصطفى عبدالرحمن أفوض جامعة فيلادلفيا بتزويد نسخ من رسالتي للمكتبات أو المؤسسات أو الهيئات أو الأشخاص عند طلبها .

التوقيع:

التاريخ: ٢٠١٣/١/٧

Philadelphia University

Authorization Form

I, Yehia Moustafa Abd AL-Rahman, authorize Philadelphia University to supply copies of my Thesis to libraries or establishments or individuals upon request.

Signature:

Date: 7/1/2013

Denotational Semantics for Cloud# Language

By

Yehia Moustafa Abd AL-Rahman

Supervisor

Dr. Mourad Maouche


**This Thesis was submitted in Partial Fulfillment of the Requirements for the
Master's Degree in Computer Science.**

Deanship of Academic Research and Graduates Studies

Philadelphia University

January, 2013

Committee Decision

Successfully defended and approved on7/1/2013.....	
Examination Committee	Signature
Dr.,Mourad Maouche....., Chairman	
Academic Rank:Associate Professor.....	
Dr.,Said AL-Ghoul....., Member.	
Academic Rank:Professor.....	
Dr.,Ismail M. Ababneh....., External Member.	
Academic Rank:Professor.....	
(Name of University:AL al-Bayt University.....)	

DEDICATION

Only until you have climbed a mountain can you look behind and see the vast distance you've covered, and remember those you've met along the way who made your trek a little easier. Now that this thesis is finally finished, after the many miles of weary travel, I look back to those who helped me turn it into reality and offer my heartfelt thanks.

Dedicated to my Mother, for her tireless support and encouragement of my Work, to my father and my sister Fatima. May their souls rest in peace.

Dedicated to my people in my homeland Syria who are struggling in their quest for freedom and might not have had the same opportunities that I had.

Eng. Yehia.M EL-Rahman

ACKNOWLEDGEMENT

Thanks go out to everyone along the way who made this work possible, especially my supervisor, Dr. Mourad Maouche for his tireless support and encouragement of my Work. Without You Dr.Mourad, nothing would have been done.

A special thank goes to Professor Dongxi Liu from CSIRO ICT Centre, Australia, to Professor Graeme Smith from the University of Queensland, Australia. Your comments were very helpful.

Also I would like to thank my brother and my sisters, Mohammed, Khadejah, Bayan and Fadia. Special thanks go to my friends and colleagues who have inspired me along the way.

Eng.Yehia.M EL-Rahman

List of Contents

Subject	Page
Committee Decision	ii
DEDICATION	iii
ACKNOWLEDGEMENT	iv
List of Contents	v
List of Figures	xi
List of Abbreviations	xii
List of Appendices	xiii
ABSTRACT	xiv
Part 1: Research Problem Definition	
Chapter One: Introduction	1
1.1 Research Context	1
1.2 Introduction to Cloud# Language	1
1.3 Research Problem	2
1.4 Motivations	3
1.5 Research Contributions	3
1.6 Research Methodology	4
1.7 Thesis Plan	5
Chapter Two: Literature Review	6
2.1 Introduction	6
2.2 Background on Modeling Languages	6
2.2.1 Classifications of Modeling Languages	7
2.2.2 The Basic Concepts in Modeling Languages: Model & Metamodel	9

2.3	Formal Semantics of Languages	9
2.3.1	A Brief Description on Formal Language Definition	10
2.3.2	Dynamic Semantics Styles	11
2.3.3	A Comparison on General Denotational Semantics Approaches	12
2.3.3.1	General Denotational Semantics Approaches of Modeling Languages	12
2.3.4	A Comparison on Different Approaches based on Translational Semantics	14
2.3.4.1	Different Approaches Based on Translational Semantics	14
2.3.5	Conclusion	16
Chapter Three: Method		18
3.1	Introduction	18
3.2	Method Description	18
3.3	Object-Z Background	19
3.4	Aspects of Object-Z	19
3.4.1	Class Union	20
3.4.2	Referencing Local Objects	20
3.4.3	Secondary attributes	20
Part 2: Research Contributions		
Chapter Four: An Abstract Syntax for Cloud#		22
4.1	Introduction	22
4.2	A BNF based Syntax for Cloud#	22
4.2.1	The Structure of Computation Unit (CUnit)	23
4.2.2	Computation Unit Definition	24
4.2.3	Computation Tasks (Comp)	24
4.2.4	Type Definition	25
4.2.5	Typing System for Cloud# Language	25

4.2.6	An Augmented and Refined Version of Cloud# Abstract Syntax	26
4.3	An Object-Z Metamodel based Syntax for Cloud#	28
4.3.1	Computation Unit (CUnit)	28
4.3.1.1	Computation Unit Declaration	28
4.3.1.2	Type Definition	29
4.3.1.3	Computation unit definition	29
4.3.1.4	Hypercall Definition	29
4.3.1.5	Hypercall	30
4.3.1.6	Computation Tasks	30
4.3.1.7	Process Expression	30
4.3.1.8	Process	31
4.3.1.9	Process Description	31
4.3.1.10	Process Body	31
4.3.2	Cloud# Actions	31
4.3.2.1	Action Sequence	32
4.3.2.2	noop Action	32
4.3.2.3	Event Statement ($E \Rightarrow A$)	32
4.3.2.4	Unit Control-Switch Statement ($[[CUnit]]$)	33
4.3.2.5	HCall Statement	33
4.3.2.6	Append Action	33
4.3.2.7	Update Action	33
4.3.3	Cloud# Expressions	34
4.3.3.1	Unary Expressions	34
4.3.3.2	Binary Expressions	35
4.4	Conclusion	36

Chapter Five: A Static Semantics Model for Cloud#	37
5.1 Introduction	37
5.2 Formal Static Semantics Definition	37
5.2.1 Computation Unit (CUnit)	37
5.2.1.1 Computation Unit Declaration	38
5.2.1.2 Hypercall Definition	38
5.2.1.3 Hypercall	39
5.2.1.4 Process Expression	39
5.2.1.5 Process Body	40
5.2.2 Cloud# Actions	40
5.2.2.1 Action Sequence	40
5.2.2.2 Event Statement ($E \Rightarrow A$)	41
5.2.2.3 Unit Control-Switch Statement ($[[\text{CUnit}]]$)	41
5.2.2.4 HCall Statement	41
5.2.2.5 Append Action	42
5.2.2.6 Update Action	42
5.2.3 Cloud# Expressions	43
5.2.3.1 Head Expression	43
5.2.3.2 Length Expression	43
5.2.3.3 ReturnAt Expression ($E[i]$)	44
5.2.3.4 LookUp Expression	44
5.3 Conclusion	45
Chapter Six: A Denotational Semantics Model for Cloud#	46
6.1 Introduction	46
6.2 Formal Denotational Semantics Definition	46

6.2.1	Computation Unit (CUnit)	47
6.2.1.1	Hypercall	49
6.2.1.2	Interleaving Concurrency	50
6.2.2	Cloud# Actions	55
6.2.2.1	Unit Control-Switch Statement ($\llbracket \text{CUnit} \rrbracket$)	56
6.2.2.2	HCall Statement	57
6.2.2.3	Event Statement ($E \Rightarrow A$)	58
6.2.2.4	Append Action	59
6.2.2.5	Update Action	59
6.2.2.6	noop Action	60
6.2.3	Cloud# Expressions	61
6.2.3.1	Head Expression	61
6.2.3.2	Length Expression	62
6.2.3.3	ReturnAt expression ($E[i]$)	62
6.2.3.4	LookUp Expression	63
6.3	Conclusion	63
Chapter Seven: Evaluation		64
7.1	Introduction	64
7.2	Background on the Formal Verification and Validation Techniques	64
7.2.1	Checking the consistency of a language	64
7.2.2	Proving some important properties about language models	65
7.3	A Case Study—Cloud# Basic Model	65
7.3.1	The Part of Computation Units	66
7.3.2	The Part of Processes	67
7.3.3	The Part of Hypercalls	68

7.4 Conclusion	69
Chapter Eight: Conclusions and Future Works	70
8.1 Conclusions	70
8.2 Future works	71
REFERENCES	72
Appendix 1: Formal Semantics Description of Cloud#	77
الملخص	86

List of Figures

Figure number	Figure Title	Page
Figure 2.1	Models and Metamodels in MDE Convention	9
Figure 2.2	Formal Language Specification	10
Figure 3.1	The General Approach of the Framework	18
Figure 4.1	The Abstract Syntax of Cloud#	23
Figure 6.1	The State-transition of a Computation Unit at run-time	47

List of Abbreviations

Acronym/Synonym	Meaning
A2S	Amazon Associates Web service
ASML	Abstract State Machine Language
BNF	Backus Naur Form
Comp	Computation Tasks
CPU	Central Processing Unit
CUnit	Computation Unit
CZT	Community Zed Tools
dom	Domain
DSM	Domain-Specific Modeling
DSMLs	Domain-Specific Modeling Languages
GPLs	General-Purpose Languages
GPMLs	General-Purpose Modeling Languages
HCall	Hypercall
I/O	Input/Output
Id	Identity
IP	Internet Protocol
MDE	Model-Driven Engineering
OCL	Object-Constraints Language
OZ	Object-Z
ran	Range
SOFL	Structured-Object-based-Formal Language
SU	Semantic Unit
UML	Unified Modeling Language
WSMO	Web Service Modeling Ontology

List of Appendices

Appendix Number	Appendix Name
Appendix 1	Formal Semantics Description of Cloud#

ABSTRACT

CloudMDE is considered as one of the most significant research areas in software development nowadays. It has attracted an increasing attention from the research community. CloudMDE aims at identifying opportunities for making Cloud Computing benefits from model-driven engineering techniques and vice versa. Cloud# language has been proposed as a way for using model-driven engineering techniques to support Cloud Computing. It is a domain-specific modeling language for modeling the infrastructure of the cloud. Cloud# is an imperative language with a textual concrete syntax. It manipulates the cloud infrastructure components as first class citizens. Furthermore, it supports concurrency and event-driven actions. Until now a BNF abstract syntax, a concrete syntax and an informal semantics description for Cloud# language are available. However, this language lacks a formal semantics definition. In this thesis, we have defined a formal denotational semantics for Cloud# language. Object-Z language has been used as a meta-language for defining the formal semantics of Cloud# in a single unified framework. That is, the abstract syntax, static and dynamic semantics of a single language construct are specified in one Object-Z class. Not only does this help the readability of the semantic, but if the language is enhanced or evolved, the required modifications can be done by minimal disruption to the existing semantics. Also it is possible to use some parts of semantics definition of one language to define another. On the other hand, the consistency checking for Cloud# language has been done using an Object-Z type-checker tool. A sample Cloud# model has been converted to the Object-Z specifications and then applied along with the existing formal denotational semantics to the type-checker. No typing errors have been found which indicates the consistency of Cloud# language.

Chapter One: Introduction

1.1 Research Context

Cloud computing and model-driven engineering (MDE) are two of the most dominant software engineering paradigms nowadays. Currently, there is a new trend to combine MDE and Cloud Computing so that they benefit from each other. The combination of these two paradigms has come up with a new research domain called CloudMDE. The first international workshop on CloudMDE happened lately in July, 2012. The aim of CloudMDE is to identify opportunities for using MDE to support the development of Cloud Computing (MDE for the Cloud) (e.g. Cloud# language (Liu and Zic, 2011), CloudML language (Goncalves et al, 2011), etc), as well as opportunities for using cloud infrastructure to enable MDE in new and novel ways (MDE in the Cloud) (e.g. Model as a Service (Maas) (Bruneli et al, 2010)). This research work is concerned with MDE for the Cloud.

1.2 Introduction to Cloud# Language

Cloud# is a domain-specific modeling language that has been designed to model the infrastructure of the Cloud. It is an imperative language with a textual concrete syntax. Cloud# manipulates the cloud infrastructure components as first class citizens. Furthermore, it supports concurrency and event-driven actions (Liu and Zic, 2011).

Cloud# can be viewed as an approximate form of very high-level programming languages, that is, a language above the current high-level general-purpose programming languages (GPLs). The term approximate form is used because we do not consider that languages like Cloud# can completely replace or mask out GPLs. There may be algorithmic elements that should not be modelled in a more abstract way like statements in GPLs.

The term very high-level is used to reflect our view that models are more abstract and compact than implementations expressed in a GPL. For instance, technical details related to efficient implementations of complex data structures (i.e. Computation unit, Resource,

Configuration, etc) and domain-specific functionality (i.e. event-driven actions) can be added through intelligent code generators and therefore need not be present in the model used to generate the code (Rumpe and France, 2011).

The Cloud# language has several special syntactic features for modeling the cloud infrastructure. First, the main syntactic construct in Cloud# is the computation unit, which represent either virtual machines or the cloud itself. Virtual machines can even model a composed cloud by defining a cloud that contains other clouds as sub computation units. Second, Cloud# can express different privilege levels of computations. This feature is necessary since the software comprising a cloud (i.e. virtual machine monitors, operating systems in virtual machines) always runs in different privilege levels (or CPU rings (Barham et al, 2003)).

Third, Cloud# allows different directions of control and data transfer between computation units of different privileges. In one direction, a low privileged unit can pass control and data to a high privileged unit by invoking a set of calls (i.e. hypercalls) defined in the high privileged unit. In the other direction, a high privileged unit can communicate with a lower one by running it directly (when scheduling), or accessing its state configuration or resources directly. Forth, there are no fixed resources in Cloud#. This feature is useful since different clouds may provide different resources (Liu and Zic, 2011).

1.3 Research Problem

Domain-specific modeling languages (DSMLs) play a cornerstone role in model-driven software development. They offer, through appropriate notations and abstractions, expressive power focused on, and usually restricted to a particular problem domain. DSMLs are usually defined in terms of their abstract and concrete syntax. This allows the rapid development of the language and some associated tools (i.e. editors), but does not allow the representation of their behavioral semantics (Andova et al, 2011).

Current domain-specific modeling (DSM) approaches have mainly focused on the syntactic (i.e. structural) aspects of DSMLs. Explicit and precise specification of the behavioral semantics of models has not received much attention by the DSM community

until recently, despite the fact that this creates a possibility for semantic mismatch between design models and modeling languages of analysis tools (Bryant et al, 2011).

This research work focuses on Cloud# language which is a recent domain specific language for modeling the infrastructure of the cloud. Until now a BNF syntax, a concrete syntax and an informal description of the semantics of Cloud# language are available. However this language lacks a formal semantics definition (Liu and Zic, 2011).

1.4 Motivations

Much of the success of MDE is dependent on the descriptive power of domain-specific modeling languages (DSMLs). One of the current challenges of adopting a DSML is the lack of a precise description of its semantics (Bryant et al, 2011). A DSML must have a precise meaning to be considered trustworthy. Without a precise specification of a language, it is hard to rigorously validate a model against the system being modeled, or to state precisely what a given analysis result really means. The design of the modelling language itself will be subjected to conceptual errors and irregularities. There will be no sound basis for developing tools (i.e. compilers, analyzers, etc) for the language, for verifying a software implementation of the DSML or for developing accurate user documentation (Wang et al, 2012).

Cloud# Modelers have had to rely heavily on English-language documentation (the informal description of the language semantics) to understand the language and interpret its models. However, this use of natural language is ambiguous and it also may have redundancy and sometimes contradictions in the information provided. To support a common understanding, and facilitate standardization for Cloud#, a formal semantics of its language is highly recommended.

1.5 Research Contributions

The research Contributions are listed below. A description of each contribution is given.

- ✓ **Provide Cloud# language with static and dynamic formal semantics:** the Object-Z approach is used to define the abstract syntax and static and dynamic semantics of the Cloud# language in a single unified framework. A consequence is that the

semantics of a language can be readily extended when the language is enhanced. Furthermore, it is also possible with this approach to reuse parts of the semantics specification of one language to define another.

- ✓ **Check the soundness of the Cloud# language by adopting usual techniques for reasoning and verifications:** an example model of Cloud# language has been first modeled in Object-Z language and then loaded along with the formal semantics descriptions of Cloud# into the Community Zed Tool (CZT) type checker. Type checking has been done to check to consistency of the language.

1.6 Research Methodology

An analytical research methodology based on mathematics and proof techniques has been adopted to conduct this research work. Our starting point is the BNF abstract syntax of Cloud# language, the informal semantics definition written in natural language and some example models that represent the actual concrete syntax of the language. The ultimate goal of this research work is to define a formal static and dynamic semantics definition for Cloud# language. Furthermore, this research work aims at checking the soundness of Cloud#. The research methodology can be summarized in the following steps:

- ✓ Be sure that the authors of the Cloud# language in (Liu and Zic, 2011) have completely described the Cloud# BNF abstract syntax by checking the agreement between the abstract and concrete syntax of Cloud# (through example models). Some additions/modifications to the abstract syntax have been suggested in this step.
- ✓ Select an appropriate approach for defining the denotational semantics of Cloud# language based on a set of suitable criteria.
- ✓ Define the static and dynamic semantics for Cloud# respectively using Object-Z formal language.
- ✓ Check the soundness of Cloud# language by adopting usual techniques for reasoning and verification.
- ✓ Evaluate and conclude our work.

1.7 Thesis Plan

The organization of this thesis proceeds as follows: chapter 2 presents an overview on the state of art research in modeling languages and formal semantics to develop a deep understanding of the key concepts in modeling languages and formal semantics techniques. Chapter 3 presents the Object-Z approach for formal semantics definition. It also highlights the main motivations for using this approach to define the formal semantics of the Cloud# language. Chapter 4 presents the abstract syntax of Cloud# as a BNF- based syntax and also as an Object-Z metamodel based syntax. It also presents the main modifications that should be made to the syntax of Cloud# to make it ready for formal semantics definition. Chapter 5 and 6 investigate the formalization of the static and dynamic semantics of Cloud# language respectively. Chapter 7 presents how we evaluate our work. And finally, Chapter 8 presents the conclusions and our future work.

Chapter Two: Literature Review

2.1 Introduction

This chapter presents an overview of the state of art research in modeling languages and formal semantics. The purpose of the survey is to develop a deep understanding of the key concepts in modeling languages and formal semantics techniques. It also presents two comparative studies on different techniques for defining formal semantics of modeling languages. These comparisons help in selecting the appropriate approach to conduct this research work.

The first part of this chapter presents a Background on modeling languages in terms of their classifications and their main concepts. The last part presents the main concepts in formal semantics and different techniques for defining semantics of modeling languages.

2.2 Background on Modeling Languages

Modeling languages have become an effective approach to address the increasing complexity of software system by raising the level of abstraction from programming languages. They are used to describe the system architecture, specify the structure and behaviour of the system, and document the system (Cho et al, 2011). The main purpose of modeling languages is to describe and represent knowledge or information in a high level of abstraction and in a structured way. They are used to describe different systems and domains, ranging from Software Engineering (Ji et al, 2011), to Computer Engineering (Kos et al, 2011), to Telecommunications (Adamis et al, 2005), through Business World (Rodríguez et al, 2011), among others. For instance, in software engineering, they can be used to describe system requirements (Requirement languages), system architectures (Architecture Description languages), and system implementations (Programming languages).

This section introduces different classifications of modeling languages according to some criteria and finally discusses the main concepts (i.e. modeling and metamodeling) that play a cornerstone in modeling languages.

2.2.1 Classifications of Modeling Languages

Modeling languages can be classified with respect to their purposes as General-Purpose Modeling Languages (GPMLs) and Domain-Specific Modeling Languages (DSMLs) (Cho et al, 2011), to their execution styles as Declarative Modeling Languages and Imperative Modeling Languages (Pichler et al, 2012), or to their concrete syntax as textual modeling languages and graphical modeling languages (Engelen and van den Brand, 2010). This section discusses these different classifications and comments them briefly.

✓ General-Purpose Modeling Languages (GPMLs)

GPMLs are one type of modeling languages that are used for a wide variety of purposes across a broad range of domains. For instance, UML (Unified Modeling Language), as a GPML, may be used for modeling Business processes (Rodríguez et al, 2011), database design (Ma et al, 2012), and software engineering (Ji et al, 2011).

The major disadvantage of GPMLs is their complexity (Cho et al, 2011). They offer many constructs and some of them may be hard to be understood or used by non-specialists. They also don't allow describing some specific domain needs in an accurate way.

✓ Domain-Specific Modeling Languages (DSMLs)

A DSML is a language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to a particular problem domain (Andova et al, 2011). It enables domain experts to develop accurate models using concepts in their own domain, rather than concepts provided by existing formalisms, which typically do not provide the required or correct abstractions. Domain-specific modeling has become a new trend in software development because it assists domain experts in focusing on the level of abstraction relevant to their problem space (Engine et al, 2010).

✓ Imperative and Declarative Modeling Languages

The difference between imperative and declarative languages also appears in computer programming. Imperative programming implies “say how to do something” (Pichler et al,

2012), whereas declarative programming implies to “say what is required and let the system determine how to achieve it” (Pichler et al, 2012).

Similar to imperative programming, imperative modeling languages implies “inside-to-outside” (Pichler et al, 2012) approach. Primarily, this consists in specifying the procedure of how the work has to be done. In contrast to Imperative languages, Declarative modeling is referred to as an “outside-to-inside” (Pichler et al, 2012), This mean that declarative languages do not specify the procedure in advance, instead of determining how the process has to work exactly, only its essential characteristics are described.

✓ Textual and graphical modeling languages

The appearance of a language is defined by means of its syntax. In the language driven approach, the constructs of the language are related to the concepts that have been identified in the domain space. With respect to the syntax, a language, in general, can be classified as a textual language, a graphical language, or a combination of the two approaches (Engelen and van den Brand, 2010).

Graphical or visual languages became more and more popular with the advent of model driven engineering techniques. They have several benefits over textual language, such as the ability to express complex relations in a more intuitive fashion. Graphical syntax may seem capable of expressing more than the textual syntax. They are mostly convenient of documenting specification and communicating solutions to various interest groups. However, editing graphical representations can be cumbersome (Kiel and Schneider, 2011).

Research has shown that graphical languages are not superior to textual languages and the both type of languages have their benefits. For instance there are cases where textual languages are more appropriate because of their clear structure (reading from left to right, from top to bottom) and the tools that can handle textual artifacts are widely spread and very mature. Further reasons for preferring textual languages to graphical ones are the speed of creation and suitability for editing and versioning (Kiel and Schneider, 2011).

2.2.2 The Basic Concepts in Modeling Languages: Model & Metamodel

A model is an abstraction of a part of the reality for a specific purpose, and is expressed in a modeling language. The structure of the modeling language itself (i.e. the abstract syntax) is given by another model called metamodel. So, the metamodel is a model about models and any model written in a language must conform to the metamodel (i.e. the Abstract syntax) of that language. For instance, Java program conforms to the Java grammar. Models and metamodels can be placed in layers, in MDE convention the reality is in layer M0, models that represent the reality are in layer M1 and the metamodels of those models are in layer M2 as shown in figure (2.1) below (Wolterink, 2009).

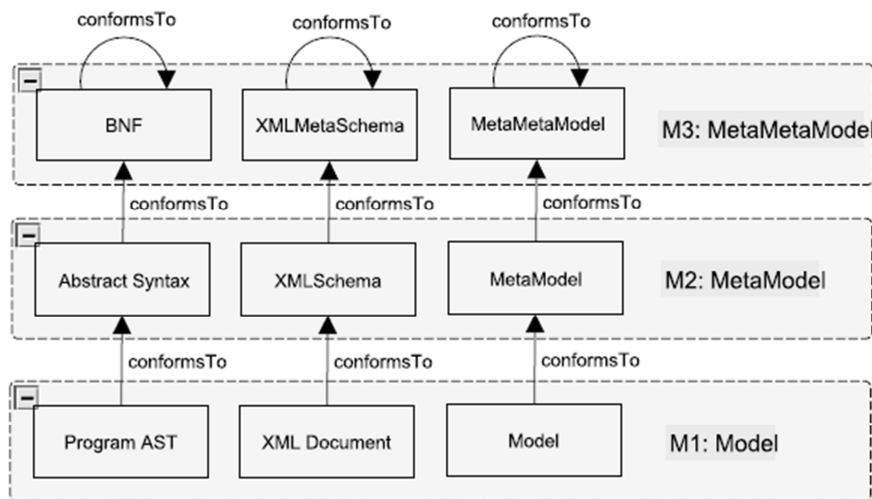


Figure 2.1: Models and Metamodels in MDE Convention

Many different definitions of metamodel may be found, but in the context of this thesis, we will consider metamodels as the specification of the abstract syntax of a modeling language (i.e. language concepts, properties on those concepts, and the existing relations between these concepts).

2.3 Formal Semantics of Languages

Language definition deals with defining the structure of the language. However, the meaning of those structures must also be defined. This meaning is defined by the semantics of the language. The semantics of a language can be defined in different ways. This section explains what the semantics are, and we take a look at different styles for

specifying dynamic semantics. Also two comparisons on different approaches for defining the semantics of modeling languages will be presented.

2.3.1 A Brief Description on Formal Language Definition

A language can be formally defined as a 5-tuple $L = \langle C, A, S, MS, MC \rangle$ as shown below, where (C) is the Concrete syntax, (A) is the abstract syntax, (MS) is the semantics mapping, and (MC) is the syntactic mapping. The syntax of a language consists of three parts: a concrete syntax which defines the specific constructs and notations that are used to express models; these models can be represented as graphical, textual, or mixed. An abstract syntax which defines the language concepts, their relationships and well-formedness rules in the language, and a syntactic mapping $MC : C \rightarrow A$, which relates the syntactic constructs into the elements in the abstract syntax.

On the other hand, language's semantics consist of two parts: a Semantic Domain (S) which explains the meaning of the language models in some formal, mathematical framework, and a Semantic Mapping $MS : A \rightarrow S$ which relates the syntactic concepts to their meaning in the semantics domain (Chen et al, 2005).

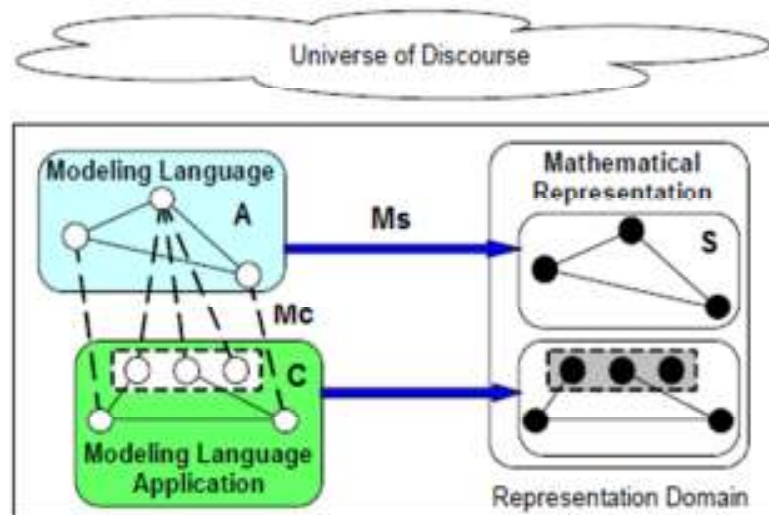


Figure 2.2: Formal Language Specification

The semantic domain invariably distinguishes between two fundamental features: static semantics which express some language constraints that are hard to be defined with BNF syntax and dynamic semantics which are used to describe the meaning and behavior of the language constructs (Wang et al, 2012).

2.3.2 Dynamic Semantics Styles

There are several styles for specifying dynamic semantics. Some of these styles are axiomatic semantics, denotational semantics, and operational semantics. These styles are discussed briefly below.

Denotational semantics: The meanings are modeled by mathematical functions that represent the effect of executing the constructs. Thus, only the effect is of interest, not how it is obtained. Denotational semantics is also called extensional semantics, because only the “extension”—the visible relation between input and output data—matters. For example, two differently coded programs that both compute factorial have the same denotational semantics (Schmidt, 2012).

Operational semantics: The meaning of a construct is specified by the sequences of computation steps that result when the construct is executed on a machine. In particular, it is of interest how the effect of a computation is produced. Operational semantics is also called intensional semantics, because the sequence of internal computation steps (the “intension”) is most important. Thus, two differently coded programs of factorial have different operational semantics (Schmidt, 2012).

Axiomatic semantics: specific properties of the effect of executing the constructs are expressed as assertions or logical propositions. Thus there may be aspects of the execution that are ignored (Schmidt, 2012).

However, it should be noted that these three styles of semantics are not rival, but are different techniques appropriate for different purposes (i.e. operational semantics are used when implementing compilers for languages, denotational semantics are used for analogous reasoning of languages (easier to understand and reason about the

mathematical representation of a language rather than the languages' constructions themselves (Naumenko et al, 2003)), while axiomatic semantics are used to prove properties of programs rather than their meanings). In fact, they are complementary and highly dependent on each other (Lester et al, 2011).

2.3.3 A Comparison on General Denotational Semantics Approaches

The problem of defining the denotational semantics of a modeling language is not new. Some considerable amount of research has already been done. This has led to a number of approaches for defining the denotational semantics. In all approaches there is a mapping from the modeling language to a semantic domain. However, this mapping is done in different ways. This section presents the main general approaches for defining the denotational semantics of modeling languages and investigates each of them according to some criteria.

2.3.3.1 General Denotational Semantics Approaches of Modeling Languages

There are two general approaches for defining the denotational semantics of modeling languages: the translational semantics approach (Bryant et al, 2011) and the traditional denotational semantics approach (Lester et al, 2011). This section investigates each approach with respect to some criteria and recommends the most appropriate one.

✓ Translational Semantics Approach

This approach consists in mapping the abstract syntax of a modeling language into the abstract syntax of an existing formal specification language with well-defined and understood semantics (Bryant et al, 2011).

The main advantage of this approach is that the existing tools (i.e. model checker, test-case generator, etc) of the target formal language may be reused (no need to build specific tools for the DSML language). However, it is very challenging to correctly map the constructs of the DSML into the constructs of the formal language, because there is no direct mapping between the source and target languages (different level of abstraction).

Another challenge is how to map the execution results (i.e. error messages, debugging traces) back into the DSML in a meaningful manner, so that the domain expert who uses the DSML understands these messages.

However, Translational semantics approach supports different styles of dynamic semantics (i.e. denotational, operational and a mix-approach). The authors in (Wang et al, 2012) have proposed a formal denotational semantics model using Object-Z language for the semantic web service ontology (WSMO). The work of (Hahn, 2008) uses the combination of Object-Z and timed-refinement calculus languages to give a consistent formal denotational and operational semantics model of a DSML for multiagent systems. Furthermore, the author in (Rusu, 2011) has proposed a formal operational semantics model using Maude language for a DSML language called xSPEM. Based on such a semantics definition, simulation, reachability and model-checking analysis tools can be generated.

✓ **Traditional Denotational Semantics Approach**

This approach consists in mapping each syntactic construct in a language directly into its mathematical meaning by using mathematical objects (i.e., Algebra, Functions, Sets, etc.). For instance, the effect of a sequence of statements separated by ‘;’ is the functional composition of the effects of the individual statement (Lester et al, 2011).

The difference between this approach and translational semantics approach is that translational semantics approach maps target language constructs into a high-level formal constructs, classes, while this approach maps target language constructs into primitive formal constructs, sets and functions. As the language being specified grows larger (enhanced or evolved), it becomes very difficult to understand or extend these specifications, because of bad structuring (i.e., the abstract syntax, static and dynamic semantics are specified separately). Well, if the language is enhanced or evolved, the modification to the already existing semantic will be very costly. Furthermore, the traditional denotational semantics approach assumes that the modeler has a very strong mathematical background (Dong, 2005).

2.3.4 A Comparison on Different Approaches based on Translational Semantics

Translational semantics approach is considered as the most common way for defining the semantics of modeling languages. Some considerable amount of research has already been done based on the translational semantics approach. This section presents the main approaches based on the translational semantics and investigates each of them according to some criteria.

2.3.4.1 Different Approaches Based on Translational Semantics

There are two main different approaches based on translational semantics: the approach that is based on a formal meta-modeling language (Wang et al, 2012), (Rusu, 2011), and the approach using Semantic unit (semantic anchoring) (Chen et al, 2005). This section investigates each approach with respect to its: modularity, applicability, reusability and extendibility.

✓ **Translational Semantics Based on a Formal Meta-modeling Language**

This approach maps the syntactic construct of the modeling language being defined into a formal meta-modeling language with well-defined and understood semantics. The meaning is given according to the semantics of the formal meta-modeling language. There are different languages that are used as meta-languages for defining the semantics of other languages. For instance, Object-Z (**OZ**) is used as a meta-language to provide a formal specification for all the aspects of a language (i.e. the abstract syntax, the static and dynamic semantics) in a single unified framework, so that the semantics of a language can be more consistently defined and revised as the language evolves (Wang et al, 2012).

Object-Z is an extension of the Z formal specification language to accommodate object orientation and scale with large specifications. Object-Z is used as a modeling language and also as a meta-language. It has been used to define the denotational semantics for textual languages (Wang et al, 2012), graphical languages (Hahn, 2008) and also for languages that support mixed notations such as SOFL (Dong, 2005).

The main advantage of using OZ is modularity. That is, the abstract syntax, static semantics and dynamic semantics of an individual construct are typically defined in one object-z class. Not only does this help the readability of the semantics, but also if the language is enhanced or extended, the corresponding semantic modifications can be captured by minimal disruption of the existing semantics. Furthermore, it is also possible with this approach to reuse parts of the semantics specification of one language to define another (Wang et al, 2012).

On the other hand, Maude which is a high level language that supports modularity and reusability is also used as a meta-language for defining the semantics of other languages. Maude is an efficient rewriting engine that integrates functional programming with rewriting logic and provides meta-language capabilities. Because of the facilities and the capabilities of Maude, it is used as a notation and a semantic framework for specifying the semantics of modeling languages. Furthermore, the semantics rules defined by Maude can be modified easily without modifying the overall semantics rules (Rusu, 2011).

Maude has been used to specify the semantics for textual and graphical languages (Rivera et al, 2009). The formal semantics of a DSML language are specified in Maude in terms of rewrite rules. The Maude rewrite rules are based on rewriting-theory and are specified in a textual form.

✓ **Translational Semantics Based on Semantic Unit**

This approach is also called semantic anchoring and it uses the well-known abstract state machines (ASM) formalism to define the semantics. It consists of specifying transformation rules between the abstract syntax of the main DSML language which was defined in a UML/OCL-based metamodel and that of a selected Semantic unit (SU) that has been defined in the Abstract State Machine Language (ASML) (Chen et al, 2005).

Semantic anchoring approach is more applicable for graphical languages in which their abstract syntax and static semantics have been defined in UML/OCL-based metamodels. One advantage of this approach is that the already existing semantic units (USs) can be re-used to easily specify the semantics of other languages.

However, this is not always possible. For instance, in heterogeneous systems, the semantics is not always fully captured by a predefined SU, if the semantics is specified from scratch it is not only expensive but we lose the advantages of anchoring the semantics to a set of common and well-established SUs. This is not only losing reusability of previous efforts, but has negative consequences on our ability to relate semantics of DSMLs to each other and to guide language designers to use well understood and safe behavioral and interaction semantic as well.

Furthermore, semantic anchoring requires well understood and safe semantic units and it is not clear how to specify the language semantic from scratch when these semantic units do not yet exist (Gargantini et al, 2009). Also if the language is enhanced or extended, it is required to re-apply the semantic anchoring process once more which seems a bit costly.

2.3.5 Conclusion

Any language needs to be defined with different styles of dynamic semantics (i.e., denotational, operational, etc.). Each style of dynamic semantics has its applications. For instance, we prefer the denotational approach when reasoning about the language, while we may prefer the operational approach when implementing the language. Cloud# is a new language and still evolving. It has no ready tools, so the first step in defining the semantics of Cloud# language is to check if soundness of this language (i.e. has no conceptual or designing errors). Denotational semantics provide an environment for verification and validation.

This thesis is concerned with defining the formal denotational semantics of Cloud# language. The denotational semantics approach with Object-Z language is used to define the semantics of Cloud# language over the other approaches. In the next few paragraphs we show our justification about this selection.

Object-Z approach is one of the most interesting approaches in translational semantics. It provides a single, canonical and unambiguous specification of a language, which can be beneficial to the semantics of that language in many different ways. It provides a formal specification of all aspects of the language in a single unified framework, so that the

meaning of the language can be more consistently defined and revised as the language evolves. That is, the abstract syntax, static and dynamic semantics of a language are specified in the same Object-Z class. Not only does this help the readability of the semantics, but if the language is enhanced, the corresponding semantics modifications can be captured by minimal disruption to the existing semantics. It is also possible with this approach to reuse parts of the semantics specification of one language to define another (Wang et al, 2012).

On the other hand, the semantics of Object-Z itself is well studied. It has a fully abstract semantics (Smith, 1995, a&b). The denotational semantics (Griffiths and Rose, 1995) and axiomatic semantics (Smith, 1995, a&b) of Object-Z are closely related to Z standard work (Woodcock and Brien, 1991). Also, Object-Z provides some useful constructs, such as Class-union (Dong and Duke, 1993) which can define the polymorphic nature of language constructs effectively. Furthermore, Object-Z also provides a range of tool (i.e. model checker) which can be beneficial to the language being specified especially if that language is new and still has no ready tools. Some considerable amounts of works have been done based on the Object-Z approach such as the work in (Wang et al, 2012), (Wang et al, 2007), (Hahn, 2008), (Dong, 2005), etc.

Object-Z has been integrated with many different formal languages (i.e. timed-refinement calculus, Pi-Calculus, etc) which makes OZ promising to define all the aspects of a language (i.e. abstract syntax, static, operational and denotational semantics) in a consistent way. For instance, Object-Z has been integrated with timed-refinement calculus to provide a consistent approach that combines different styles of dynamic semantics (i.e. denotational and operational semantics) in a single unified framework (Hahn, 2008).

Chapter Three: Method

3.1 Introduction

This chapter aims to describe the selected approach that has been adopted to conduct our research work in details. So, in the first part of this chapter, we present the selected approach in more details and finally, in the second part of this chapter, we present some basic knowledge that is useful to understand the elaborated specifications.

3.2 Method Description

The Object-Z approach provides a formal specification for all the aspects of Cloud# language in a single unified framework, so that the semantics of Cloud# language can be more consistently defined and revised as the language evolves. Figure 3.1 below shows the general approach of the framework.

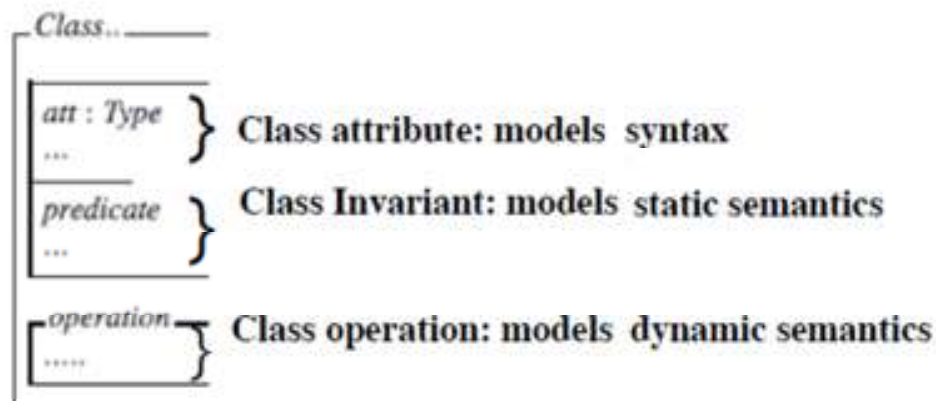


Figure 3.1: The General Approach of the Framework

The language constructs are specified as different Object-Z classes. The syntax of an individual language construct is captured by the attributes of an Object-Z class. The predicates are defined as class invariants used to capture the static semantics of the language. The class operations are used to define the denotational semantics of the language.

In this approach, a model/program in a language is modeled as an object which consists of a collection of objects. Different Object-Z classes are used to model the different language constructs. For instance, if **Expressions** are modeled by an Object-Z class, say **Exp**, and then any individual expression in a program is modeled as an instance of the Object-Z class **Exp**. In the next section we will present a brief description on the basic knowledge in Object-Z language that helps to understand the elaborated specifications.

A formal denotational semantics model for Cloud# language will be presented in this thesis in an incremental way. That is, the abstract syntax of Cloud# will be presented in Chapter 4, the static and dynamic semantics of Cloud# will be presented in Chapter 5 and 6 respectively. Because of the limitations in space, we only presented the formalization of the main syntactic constructs in Cloud# language. However, the rest of the formal description is available in the appendix at the end of this thesis.

3.3 Object-Z Background

Object-Z is an object-oriented formal specification language developed at the software verification research center at the University of Queensland. It is founded on the simple and easily understandable mathematical theory of typed sets. Object-Z is an extension of the Z formal specification language to accommodate object orientation. The main reason for this extension is to improve the clarity of large specifications through enhanced structuring (Smith, 1999).

The essential extension to **Z** given by Object-Z is the *class* construct, which groups the definition of a state schema and the definitions of its associated operations. A class is a template of objects: for each object, its states are instances of the state schema of the class and its individual state transitions conform to individual operations of the class. An object is an instance of a class and evolves according to the definitions of its class (Smith and Winter, 2012).

3.4 Aspects of Object-Z

In this section, some aspects of Object-Z that have been used in this thesis, such as class union, and object containment will be explained briefly.

3.4.1 Class Union

Class union enables the definition of a set comprising the object identities of a collection of classes. It is a more general form of polymorphism than that of the traditional polymorphism, since the classes need not be related by inheritance nor have any restrictions on their features. The expression constructs a set of object identities which is the union of the sets of identities of its constituent classes. Like set union, class union is commutative and associative.

$$\mathbf{Expression ::= Expression \cup Expression}$$

Each constituent expression of a class union is either a class union expression or a class name. We have adopted class union in this thesis, because it is a more general form of polymorphism. For more information about class union please refer to (Dong and Duke, 1993).

3.4.2 Referencing Local Objects

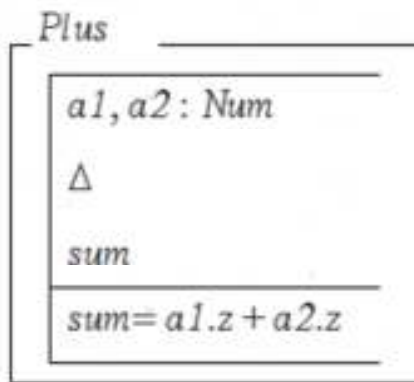
Typically, objects locally reference other objects in any object-oriented system. This facilitates the sending and receiving of messages. However, such referencing could result in a complex structure or in a cyclic pattern (i.e. inside an expression may be other expressions, but an expression cannot be inside itself).

To solve this inconsistency, a condition must be added to ensure acyclic pattern. In Object-Z, the subscript ‘ \circledast ’ is added to some attributes in a class to ensure this condition implicitly (for more information on local object reference read (Dong et al, 1997)).

3.4.3 Secondary attributes

In Object-Z, there are two types of attributes that can be defined in a class, the primary and secondary attributes (semantic variables). Delta is used to separate these kinds of attributes. The values of the primary variables determine the state of an object. On the other hand, the values of secondary variables depends on the primary variables values. For instance, in the class below, the value of the secondary attribute sum depends on the values of the primary attributes a1 and a2. This means that the secondary attribute sum

stores run-time information about the Plus object (i.e. its value). For more information about secondary attributes read (Smith, 1999).



Chapter Four: An Abstract Syntax for Cloud#

4.1 Introduction

This chapter presents the abstract syntax of Cloud# language as a BNF based syntax and also as an Object-Z metamodel based syntax. The first part of this chapter presents the main modifications that have been made to the syntax of Cloud# to make this language ready for formal semantics definition. The second part of this chapter presents an Object-Z metamodel based syntax for Cloud#. And finally, a small conclusion is presented.

4.2 A BNF based Syntax for Cloud#

The abstract syntax of a language deals solely with the well-formedness rules that make up legal expressions in a language without any consideration given to their meaning. The meaning of a language is defined by mapping the syntactic domain into a semantic domain. So the abstract syntax must be specified precisely prior to semantics since meaning can be given only to correctly formed expressions in a language (Stuurman, 2010).

The first step in our research methodology is to make sure that the authors of Cloud# language in (Liu and Zic, 2011) have completely described the Cloud# BNF abstract syntax. This is done by studying the agreement or conformance between the abstract and concrete syntax of Cloud# language (through example models). After this study, we noticed that there are some missing parts in the syntax that must be explicitly defined so that the language becomes ready for formal semantics definition. Some modifications and additions have been proposed. Figure 4.1 below shows the only published version of Cloud# abstract syntax.

```

CUnit ::= (id, Comp, [CUnit1, ..., CUnitn], HCallDefs, Conf, Res)
Comp ::= Actions | Actions | Actions
Actions ::= Ai | Ai Actions | Var := HCall; Actions | Var := E; Actions
A ::= bl : A | goto bl | E → A | noop | if E then Actions else Actions
        for Var in E do Actions | return E | [CUnit] | HCall
E ::= Var | Con | self | {E}E' | (E1, ..., En) | E.i | [E1, ..., En] | E[i] | length(E) | head(E)
        append(E, E) | E1 Op E2
OP ::= > | = | < | + | -
HCallDefs ::= {HCall1 = Actions1, ..., HCalln = Actionsn}
HCall ::= Name(Var1, ..., Varn)
T ::= Name | (T1, ..., Tn) | [T]
Conf, Res ::= {Name1 = T1, ..., Namen = Tn}
Name, Var, Con ::= String of letters or digits

```

Figure 4.1: The Abstract Syntax of Cloud#

This section presents the main modifications and additions that have been suggested to the syntax of Cloud# language. It mainly discusses five points about this language: the structure of computation unit (CUnit), the computation unit definition, the computation tasks, the type definition, and also a typing system for Cloud# language. Finally, an augmented and refined version of the abstract syntax is presented.

4.2.1 The Structure of Computation Unit (CUnit)

The current syntax of Cloud# defines the computation unit (CUnit) as a tuple of six components (Liu and Zic, 2011). These components are defined outside the borders of the CUnit, so it becomes difficult for the compiler to distinguish between the components of different CUnits or to state where a specific CUnit begins or ends. In this section, we propose a suitable structure for any computation unit that groups the declaration of CUnit and its components in a single structure. That is, any CUnit contains a declaration which identifies the signature of its components and a body which identifies the definitions of these components. The production rules below ensure such requirements.

```

CUnit ::= CUnitDec { TypeDefinitions Components CUnitDefinitions }
CUnitDec ::= (ID, Comp, [ CUnit1, ..., CUnitn ], HCallDefs, Conf, Res )
Components ::= Res ';' Conf ';' Processes HCallDefs

```

4.2.2 Computation Unit Definition

According to the concrete syntax of Cloud# language (Liu and Zic, 2011), any new computation unit (**CUnit**) must be defined before we can use it. As shown below, there are two different Computation Units with the identities **vm1**, and **vm2** respectively.

```
(vm1, noop, [], {}, {}, VmRes)
(vm2, noop, [], {}, {}, VmRes)
```

If the modeler has the intention to use these Computation Units in other places (i.e. use them as sub-units in other Computation Units), then he must define them in advance as follows:

```
VM1 = (vm1, noop, [], {}, {}, VmRes)
VM2 = (vm2, noop, [], {}, {}, VmRes)
```

After defining Computation Units, now it is possible for the modeler to use them in other places as follows:

```
(vmm, Scheduler | Network, [VM1, VM2], HCallDefs, Conf, Res)
```

We have introduced a new BNF Production Rule for defining Computation Units as shown below:

```
CUnitDef ::= ID '=' CUnit
```

4.2.3 Computation Tasks (Comp)

The current abstract syntax of Cloud# language defines the computation tasks as an action sequence or a parallel composition of different action sequences (Liu and Zic, 2011). In fact, such description is very abstract and doesn't perfectly match of what stated in the available concrete syntax. In the concrete syntax of Cloud#, a computation task can be a noop action to indicate that there are no computations, or it can be a parallel composition of different processes. On the other hand, A process can be atomic (i.e. has a unique identifier and is described by a parallel composition of different action sequences) or it can be composite (i.e. has a unique identifier and is described by a parallel composition of different processes). The following production rules ensure these requirements.


```

Comp ::= noop | ProcessExp

ProcessExp ::= ProcessId | ProcessId '|' ProcessExp

Process ::= ProcessId '=' ProcessDescription

ProcessDescription ::= ProcBody | ProcessExp

ProcBody ::= '{' Actions | Actions '|' ProcBody '}'

```

4.2.4 Type Definition

Cloud# language gives the ability for its users to introduce new data-types depending on the already existing data-types. This allows modeling more complex structures and reusing these structures throughout Cloud# models. The type definition consists of an identity (**id**) and a specific type. Two production rules have been added to the abstract syntax of Cloud# language for type definition as follows:

```

TypeDefinitions ::=  $\epsilon$  | TypeDef ';' TypeDefinitions

TypeDef ::= ID '=' Type

```

4.2.5 Typing System for Cloud# Language

No clear typing system appears in the current syntax of Cloud# language. In this section we suggest a clear typing system which includes all the required data-types. These data-types should be available to ensure well-typing models.

Data-types can be classified as follows:

- ✓ **Pre-defined types:** integer, string and Boolean.
- ✓ **User-defined types:** Base, Tuple, Record and List.
- ✓ **Void type:** this type is used for the operations that don't return values (i.e. Write () is a hypercall that returns nothing).

The BNF production rules bellow present the main types that should be available in Cloud#.

```

T ::= PreDefType | UserDefType
PreDefType ::= integer | string | boolean | VoidType
UserDefType ::= BaseType | '(' T1 ';' ... ';' Tn ')' | '[' T ']' | '{' ID1 '=' T1 ';' ... ';' IDn '=' Tn '}'

```

Note: Base-Type is a user-defined type and it can be anything the user defines (i.e. it can be IPAddress to mean the type of IP address).

4.2.6 An augmented and refined Version of Cloud# Abstract Syntax

```

CUnit ::= CUnitDec { TypeDefinitions Components CUnitDefinitions }
CUnitDec ::= (ID, Comp, [ CUnit1, ..., CUnitn ], HCallDefs, Conf, Res )
TypeDefinitions ::= ε | TypeDef ';' TypeDefinitions
TypeDef ::= ID '=' Type
Components ::= Res ';' Conf ';' Processes HCallDefs
Conf, Res ::= '{' ID1 '=' T1 ';' ... ';' IDn '=' Tn '}'
Processes ::= ε | Process Processes
HcallDefs ::= ID '=' '{' HCall1 ';' ... ';' HCalln '}'
CUnitDefinitions ::= ε | ID '=' CUnit ';' CUnitDefinitions

```

```

Comp ::= noop | ProcessExp
ProcessExp ::= ProcessId | ProcessId '|' ProcessExp
Process ::= ProcessId '=' ProcessDescription
ProcessDescription ::= ProcBody | ProcessExp
ProcBody ::= '{' Actions | Actions '|' ProcBody '}'

```

```

Lbl, BaseType, Var ::= ID

```

Actions	 ::= seqAction
seqAction	 ::= ϵ A ';' seqAction
A	 ::= noop Var ':=' SymbolicVal lbl ':' A goto lbl E \Rightarrow A if E then '{ Actions }' else '{ Actions }' for Var in E do '{ Actions }' return E ';' '.' CUnit '.' HCallStmt 'Update(' E ',' E ',' E ')' 'append(' E ',' E ')'
SymbolicVal	 ::= Expression HCallStmt
HCallStmt	 ::= ID '(' Exp ₁ ',' ... ',' Exp _n ')'

E	 ::= Var Con self UnaryExp BinaryExp DotExp
UnaryExp	 ::= 'length(' E ') 'head(' E ')'
BinaryExp	 ::= E1 OP E2 '{ E } E' E '[' i ']' 'LookUP(' E ',' E ')'
OP	 ::= > = < + -
DotExpression	 ::= ID '.' ID ID '.' DotExpression

HCall	 ::= ID '(' Var ₁ ':' T ₁ ',' ... ',' Var _n ':' T _n ') ':' T '=' { Actions }
--------------	--

T	 ::= PreDefType UserDefType
PreDefType	 ::= integer string boolean VoidType
UserDefType	 ::= BaseType '(' T ₁ ',' ... ',' T _n ') '[' T ']' '{ ID ₁ '=' T ₁ ',' ... ',' ID _n '=' T _n }'
Con	 ::= String of letters or digits
ID	 ::= Alphabet Alphabet ID
Alphabet	 ::= 'a' 'b' 'c' ... 'z'

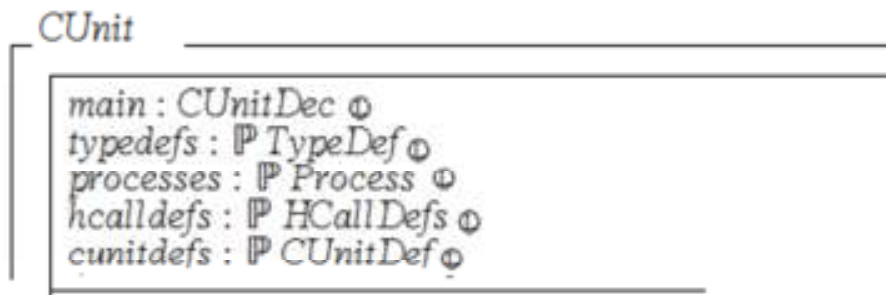
4.3 An Object-Z Metamodel based Syntax for Cloud#

As mentioned in chapter 3, the Object-Z approach which is used for defining the formal semantics of Cloud# language is based on translational semantics. That is, the abstract syntax of Cloud# must be translated into the abstract syntax of Object-Z language prior semantics definition. So, every Cloud# language construct is modeled as a class in Object-Z language. The syntax of each individual construct is captured by the attributes of its class. The result will be an Object-Z metamodel based syntax for Cloud# language.

This part describes the formalization of the abstract syntax of individual constructs in Cloud# language. The Abstract syntax for Cloud# Computation unit with its components, Actions, and expressions will be described respectively.

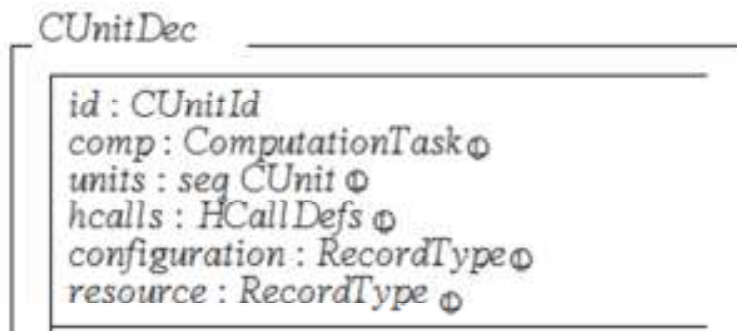
4.3.1 Computation Unit (CUnit)

Computation Unit (**CUnit**) is the main syntactic construct in Cloud# language. It is dedicated to model clouds, virtual machines or operating systems, etc. A computation unit is represented with a declaration and the definitions of CUnit components (i.e. type definitions, processes, hypercall definitions, and CUnits definitions). These components are all modeled as attributes in an Object-Z class named **CUnit** as follows:



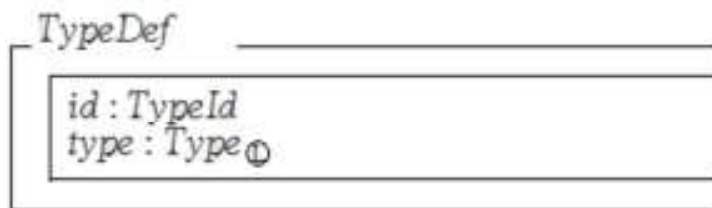
4.3.1.1 Computation Unit Declaration

The declaration of CUnit is represented as a Tuple of six components: a unique identifier (**id**), a computation task (**Comp**), a list of sub computation units (**Units**), a Hypercall definition (**HcallDefs**), a configuration part (**Configuration**), and a resource part (**Resource**). The declaration can be modeled as follows:



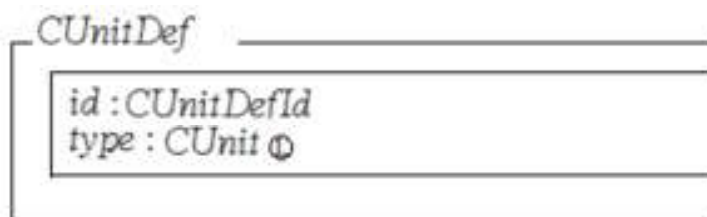
4.3.1.2 Type Definition

Syntactically, the type definition consists of an identity (**id**) and a specific type. This type can be a simple type (i.e. Boolean, integer, etc) or a composite type. It can be modeled as follows:



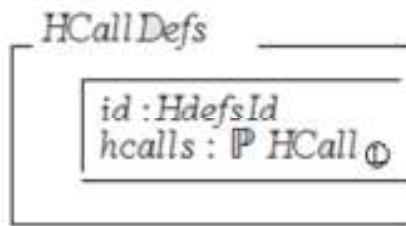
4.3.1.3 Computation unit definition

Syntactically, the Computation Unit definition consists of an identity (**id**) and a specific CUnit. It can be modeled in an Object-Z class named **CUnitDef** as follows:



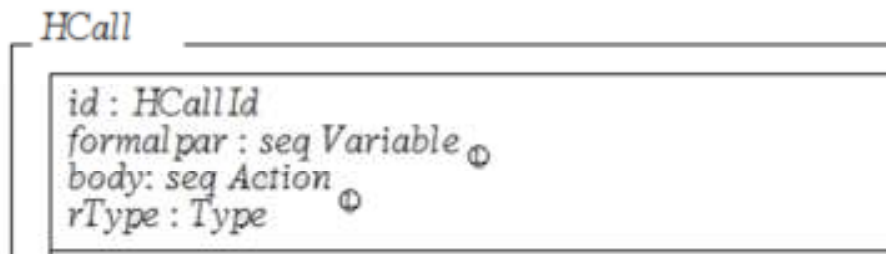
4.3.1.4 Hypercall Definition

Hypercall definition is defined with an identify (**id**) and as a set of Hypercalls which are used by the sub-CUnits to handle privileged operations. It can be modeled as follows:



4.3.1.5 Hypercall

A hypercall in Cloud# is a syntactic construct like a procedure in Pascal. Syntactically, a hypercall consists of an identity (**id**), a formal parameter list (**formalpar**), a body (**body**), and a return type (**rType**). It can be modeled as follows:



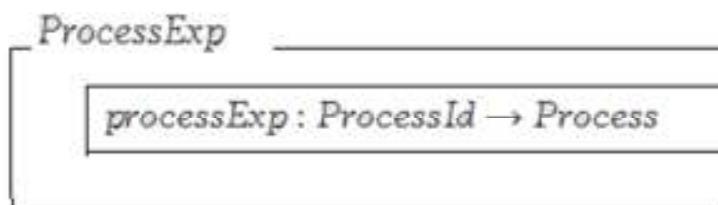
4.3.1.6 Computation Tasks

A computation task indicates the computations that a **CUnit** intends to perform, for instance, virtualization of resources and scheduling of computation tasks. It can be modeled in a class union as a noop action or a process expression.

$$\text{ComputationTask} \triangleq \text{noop} \cup \text{ProcessExp}$$

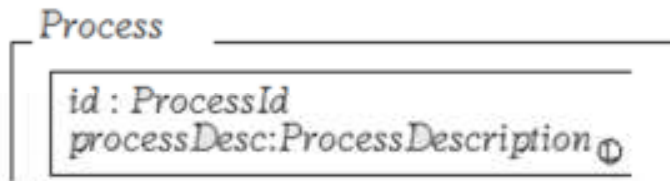
4.3.1.7 Process Expression

A Process Expression represents a set of Process-identities which refer to some processes in Cloud model. It is defined as a total function that maps every process identity in that process expression to a specific process. It can be modeled as follows:



4.3.1.8 Process

A process is defined with an identifier (**id**) and a process description (**processDesc**). It can be modeled as follows:



4.3.1.9 Process Description

Syntactically, a process description can be defined as a body (i.e. a parallel composition of different action sequences) or a process Expression (i.e. a parallel composition of different processes) and it is modeled in a class union as follows:

$$\text{ProcessDescription} \cong \text{ProcBody} \cup \text{ProcessExp}$$

4.3.1.10 Process Body

Syntactically, the Process Body consists of a parallel composition of different action sequences. It can be modeled as follows:



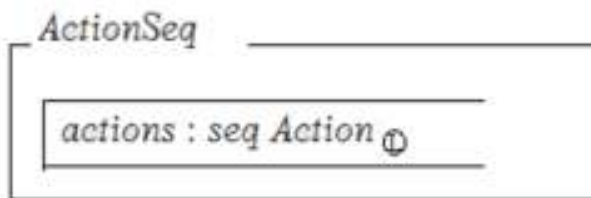
4.3.2 Cloud# Actions

Cloud# action can be a simple noop action, an assignment-statement, an if-statement, a for-statement, an event-statement, a unit control-switch statement, an HCall-statement, a return-statement, a label-statement, an Append action, and an Update action or a goto-statement. This is captured by the class-union as follows:

$$\begin{aligned} \text{Action} \quad \hat{=} \quad & \text{Assignment} \cup \text{LabelStmt} \cup \text{GotoStmt} \cup \text{EventStmt} \cup \text{IFStmt} \\ & \cup \text{ForStmt} \cup \text{ReturnStmt} \cup \text{UnitControlSwitch} \\ & \cup \text{HCallStmt} \cup \text{noop} \cup \text{Append} \cup \text{Update} \end{aligned}$$

4.3.2.1 Action Sequence

Action sequence represents a set of actions that are executing in a sequential manner. Syntactically, an action sequence is defined as a sequence of different actions. It can be modeled as follows:



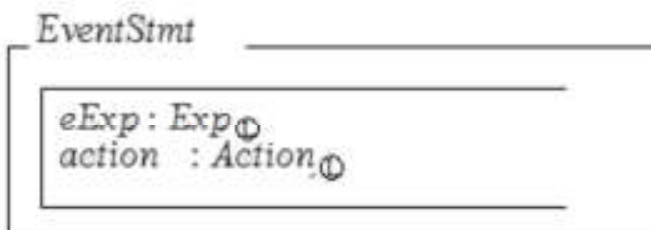
4.3.2.2 noop Action

Syntactically, a noop action is defined as an Object-Z class with empty state schema as follows:



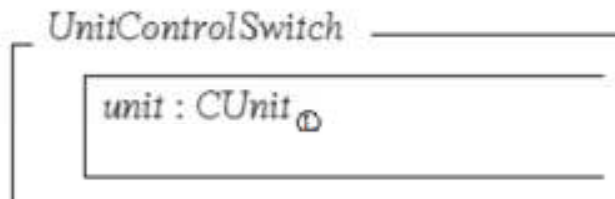
4.3.2.3 Event Statement ($E \Rightarrow A$)

Syntactically, an event statement is defined as an Object-Z class that has two attributes (expression and action) as follows:



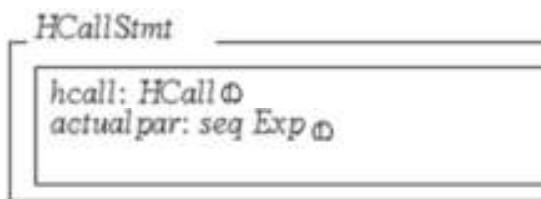
4.3.2.4 Unit Control-Switch Statement (· CUnit ·)

Syntactically, a Unit Control-Switch Statement is defined as an Object-Z class that has one attribute of a CUnit type as follows:



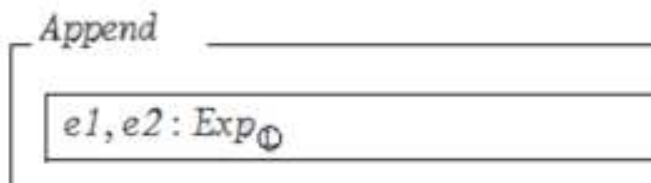
4.3.2.5 HCall Statement

Syntactically, HCall Statement is like a procedure call, it consists of a hypercall (**hcall**) and an actual parameter list (**actualpar**). It can be modeled as follows:



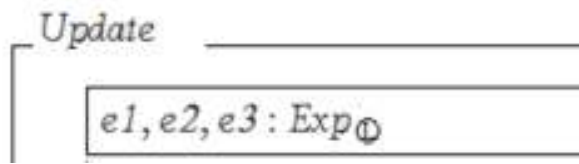
4.3.2.6 Append Action

Syntactically, the Append Action is described below in the Object-Z class named **Append**. It includes two sub- expressions (**e1** and **e2**).



4.3.2.7 Update Action

Syntactically, the Update Action is described below in the Object-Z class named **Update**. It includes three sub- expressions (**e1**, **e2** and **e3**).



4.3.3 Cloud# Expressions

In this section, Object-Z is used to specify the abstract syntax of expressions. Each expression is modeled as an object (Wang et al, 2012): a binary (unary) expression is modeled as an object with two (one) sub expressions. An expression can be a constant, variable, self-reference, unary expression, binary expression, or dot expression. On the other hand, constant can be any value except nil. Expressions can be modeled as follows:

$$Exp \triangleq Variable \cup Constant \cup Self \cup UnaryExp \cup BinaryExp \cup DotExp$$

$$Constant \triangleq IntVal \cup BoolVal \cup StrVal \cup BaseVal \cup ListVal \cup TupleVal$$

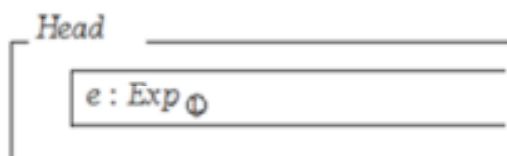
4.3.3.1 Unary Expressions

Cloud# language includes two unary expressions: the **Head** and **length** expressions. Unary Expression can be modeled as Class Union as follows:

$$UnaryExp \triangleq Head \cup Length$$

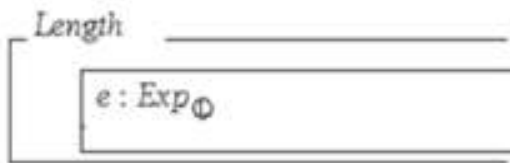
✓ Head Expression

Syntactically, the Head expression is shown below in the Object-Z class named **Head**. It includes one sub expression (**e**).



✓ Length Expression

Syntactically, the length expression is shown below in the Object-Z class named **Length**. It includes one sub expression (**e**).



4.3.3.2 Binary Expressions

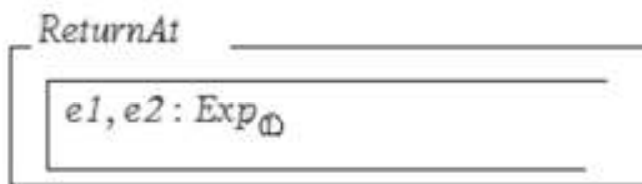
Cloud# language includes three binary expressions: arithmetic expression which includes arithmetic and logical operations (i.e. $>$, $<$, $=$, $+$, $-$), ReturnAt expression ($E[i]$) and LookUp expression. Binary expressions and arithmetic expressions can be modeled using class union as follows:

$$BinaryExp \triangleq ArithmeticExp \cup ReturnAt \cup LookUp$$

$$ArithmeticExp \triangleq Encryption \cup Less \cup More \cup Equality \cup Plus \cup Minus$$

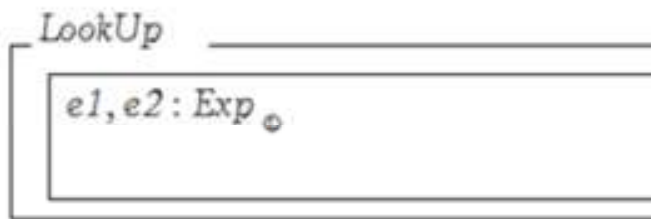
✓ ReturnAt expression ($E[i]$)

Syntactically, the ReturnAt expression is described below in the Object-Z class named **ReturnAt**. It includes two sub expressions (**e1** and **e2**).



✓ LookUp Expression

Syntactically, the LookUp expression is described below in the Object-Z class named **LookUp**. It includes two sub-expressions: **e1** which is used to locate the required element and a list **e2**.



4.4 Conclusion

In the first part of this chapter, we have presented the main modifications that have been done to the syntax of Cloud# language. These modifications made Cloud# language ready for formal semantics definition. In the second part, we built an Object-Z based metamodel for Cloud# language by translating its abstract syntax into the abstract syntax of Object-Z Language. Because of the limitations in space, we only presented the formalization of the main syntactic constructs in Cloud# language. However, the rest of the formal description is available in the appendix at the end of this thesis.

Chapter Five: A Static Semantics Model for Cloud#

5.1 Introduction

This chapter investigates the formalization of the static semantics for individual constructs in Cloud# language. The first part of this chapter presents how the Object-Z approach is used to give formal static semantics for Cloud# syntactic constructs. The static semantics will be added to the Cloud# metamodel that has been accomplished in the previous chapter. The last part of this chapter presents a small conclusion.

5.2 Formal Static Semantics Definition

Static semantics definition is the first step in the semantic analysis which manages the identifiers that are defined in the source model. It aims at detecting static semantic errors such as the redefinition of an identifier in the same scope, as well as associating identifiers with appropriate types or values. Furthermore, static semantics also manages the types of all phrases in the language. It detects type-mismatch errors such as assignment to a constant value and associates syntactically well-formed phrases with appropriate phrase types (Fisher, 2010).

In the Object-Z approach, the static semantics of Cloud# syntactic constructs are captured by class predicates or invariants. Static semantics are based solely on the abstract syntax. That is, the static semantics will be added to the Cloud# metamodel that has been accomplished in the previous chapter.

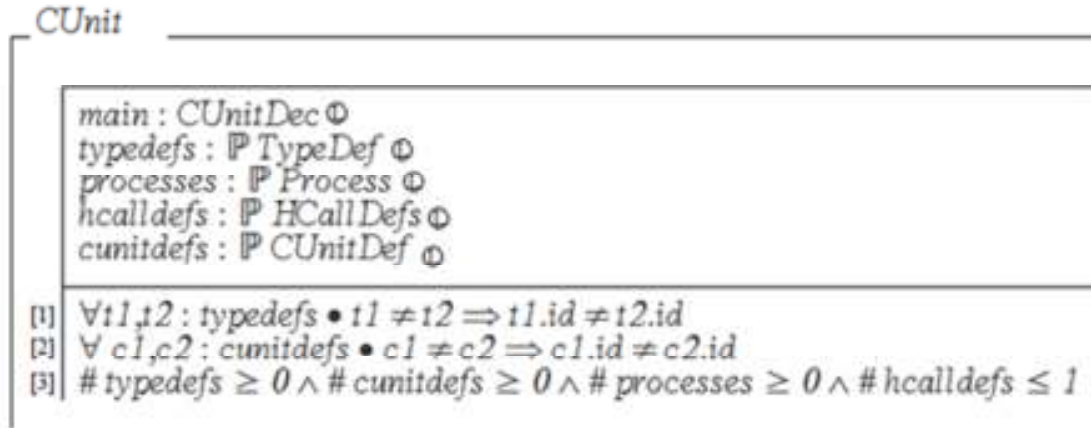
This section describes the formalization of the static semantics of individual constructs in Cloud# language. The static semantics for Cloud# Computation unit with its components, Actions, and expressions will be described respectively.

5.2.1 Computation Unit (CUnit)

The static semantics for a CUnit can be defined as follows:

- [1] Every two different defined types in a CUnit must have different identities.

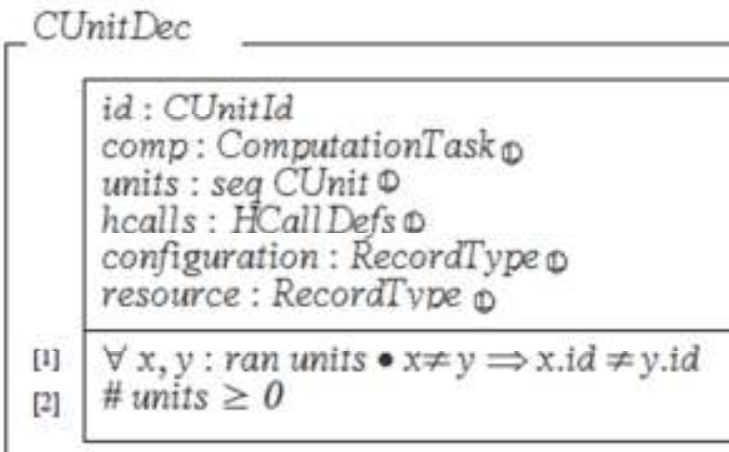
- [2] Every two different defined CUnits within the borders of the same parent CUnit must have different identities.
- [3] The number of defined types, the number of defined CUnits and the number of processes equals or more than zero. On the other hand, the number of hypercall definitions is at most equal to one.



5.2.1.1 Computation Unit Declaration

The static semantics for a Computation Unit Declaration can be defined as follows:

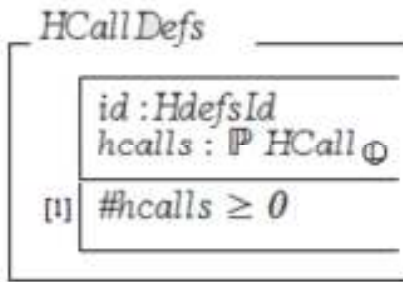
- [1] Every two different sub-units must have different identities.
- [2] The number of sub-units within the same parent CUnit is equal or more than zero.



5.2.1.2 Hypercall Definition

The static semantics for a hypercall definition are defined as follows:

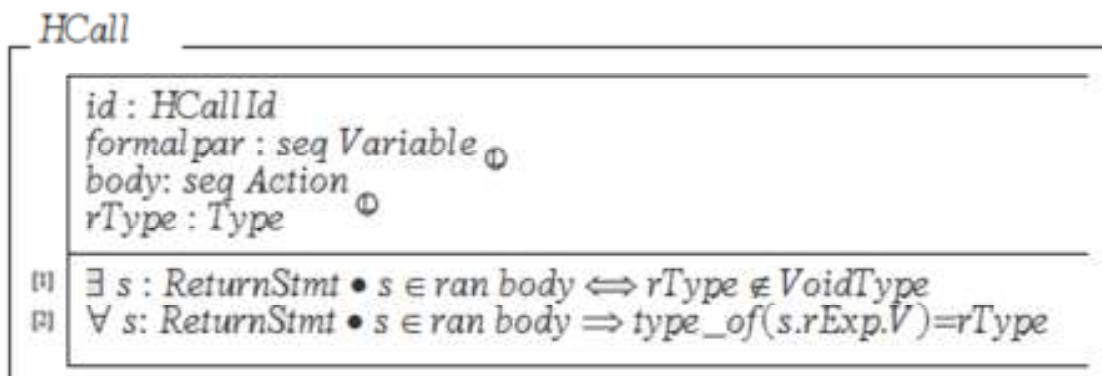
- [1] The number of Hypercalls within the same definition is equal or more than zero.



5.2.1.3 Hypercall

The static semantics for a hypercall are defined as follows:

- [1] If the return-type of an **HCall** doesn't belong to void type, then there is a return-statement (**s**) such that **s** belongs to the body (**body**) of that **HCall**.
- [2] For all return-statements that belong to the body (**body**) of an **HCall**, their types should match the return-type of that **HCall**.

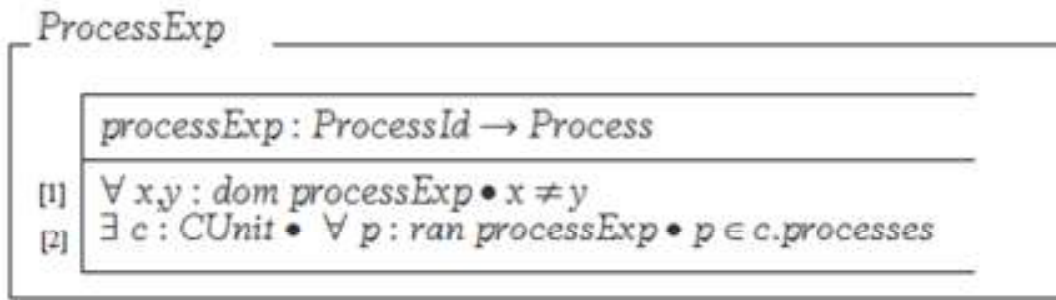


Please refer to the appendix for more information about the function (**type_of**) and the classes (**Variable**, **Type**, and **ReturnStmt**).

5.2.1.4 Process Expression

The static semantics for a Process Expression are defined as follows:

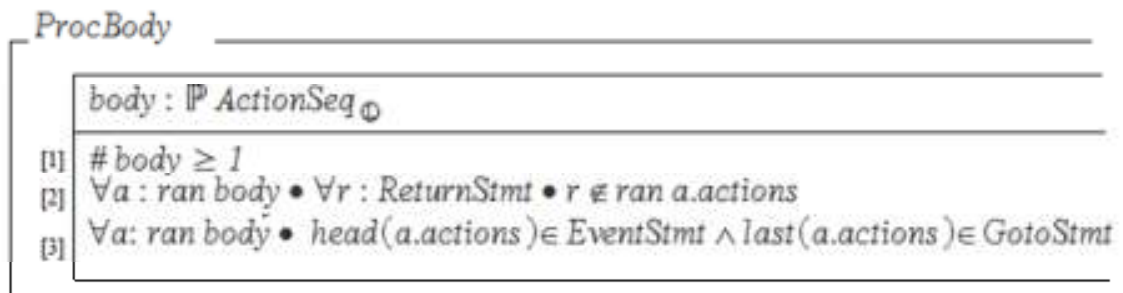
- [1] Every two ProcessIds that appear in the Process Expression must be different.
- [2] There exists a CUnit such that all the referenced-processes in the Process Expression belong to that CUnit' processes.



5.2.1.5 Process Body

The static semantics for a process body are defined as follows:

- [1] The number of action sequences within the same body of a process is equal or more than one.
- [2] The return statement is not allowed to be in the body of a process (process does not return values).
- [3] The first action in the process body belongs to Event Statement, and the last action belongs to Goto statement.



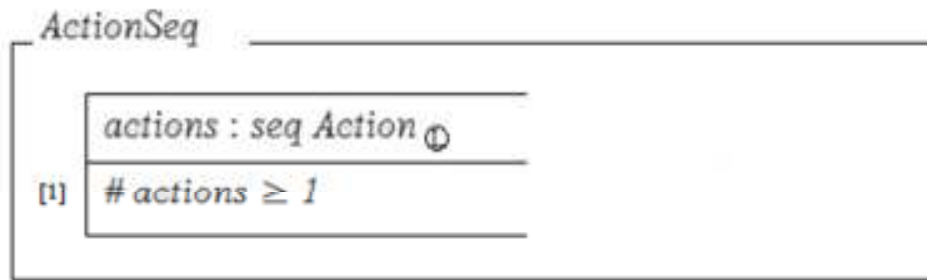
5.2.2 Cloud# Actions

This section presents the formalization of the static semantics for Cloud# actions.

5.2.2.1 Action Sequence

The static semantics for an Action Sequence are defined as follows:

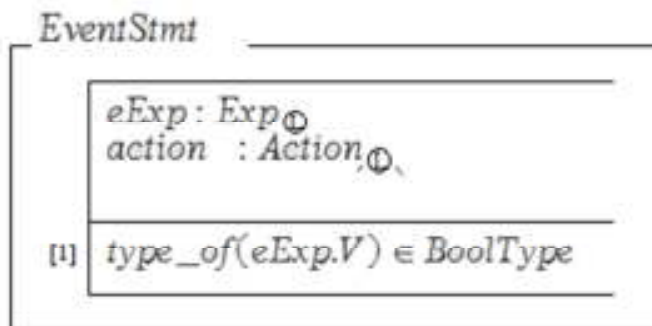
- [1] The number of actions should be equal or more than one.



5.2.2.2 Event Statement ($E \Rightarrow A$)

The static semantics for an Event Statement are defined as follows:

- [1] The value of the expression (**eExp**) should be of a Boolean Type.



5.2.2.3 Unit Control-Switch Statement ($\bullet CUnit \bullet$)

The static semantics for a Unit Control-Switch Statement are defined as follows:

- [1] The **CUnit** that appears in Unit Control-Switch Statement must be one of the sub-units of an existing computation unit (**CUnit**).



5.2.2.4 HCall Statement

The static semantics for an HCall statement are defined as follows:

- [1] The number of actual parameters in the HCall statement should be equal to the number of parameters in the HCall header.
- [2] The type of each parameter in the HCall statement should be equal to the type of its correspondence in the HCall header.

<i>HCallStmt</i>	
	$hcall: HCall \oplus$ $actual\ par: seq\ Exp \oplus$
[1]	$\# actual\ par = \# hcall.\ formal\ par$
[2]	$\forall i : dom\ actual\ par \bullet type_of(actual\ par(i).V) = hcall.\ formal\ par(i).type$

5.2.2.5 Append Action

The static semantics for an Append action are defined as follows:

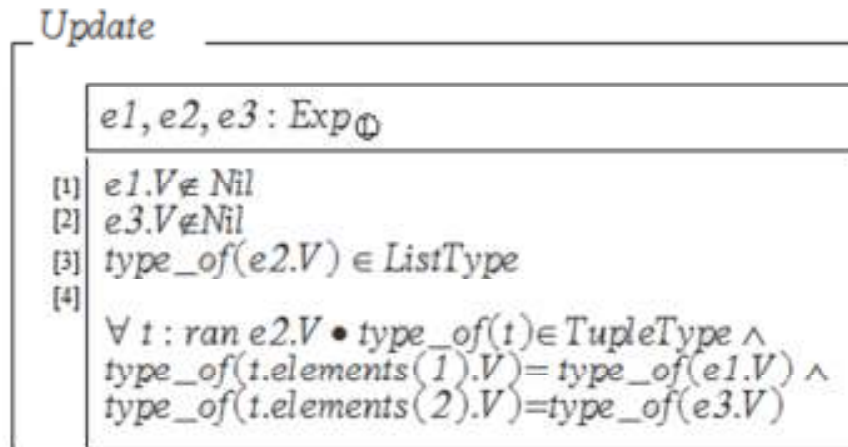
- [1] The type of the expression (**e1**) value should be of a list type.
- [2] The type of the expression (**e2**) value is the same type of the elements in the list (**e1**).

<i>Append</i>	
	$e1, e2 : Exp \oplus$
	$e1.V \notin Nil \Rightarrow$
[1]	$type_of(e1.V) \in ListType$
[2]	$\forall a : ran\ e1.V \bullet type_of(a) = type_of(e2.V)$

5.2.2.6 Update Action

The static semantics for an Update action are defined as follows:

- [1] The value of **e1** can't be **nil**.
- [2] The value of **e3** can't be **nil**.
- [3] The value of **e2** should be of a list type.
- [4] The elements in **e2** should be of a Tuple type. And for each Tuple in **e2**, the first and the second elements should have the same types of the values of **e1** and **e3** respectively.



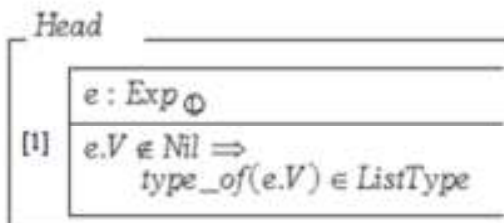
5.2.3 Cloud# Expressions

This section presents the formalization of the static semantics for Cloud# expressions.

5.2.3.1 Head Expression

The static semantics of a Head expression are defined as follows:

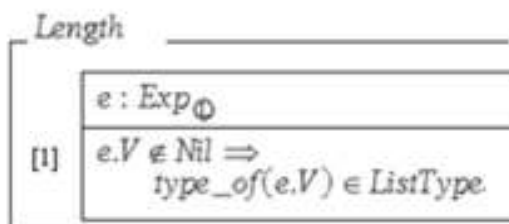
- [1] The type of the expression (**e**) value should be of a list type.



5.2.3.2 Length Expression

The static semantics of a Length expression are defined as follows:

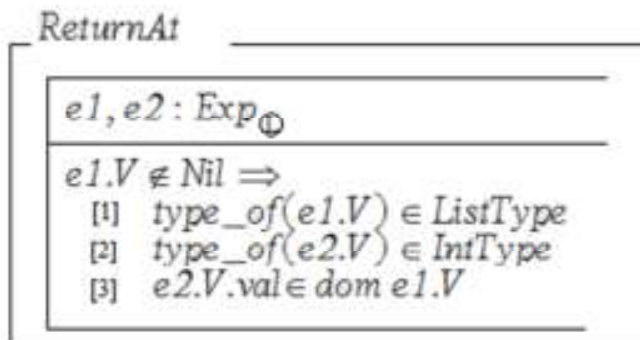
- [1] The type of the expression (**e**) value should be of a list type.



5.2.3.3 ReturnAt expression (E[i])

The static semantics for a ReturnAt expression are defined as follows:

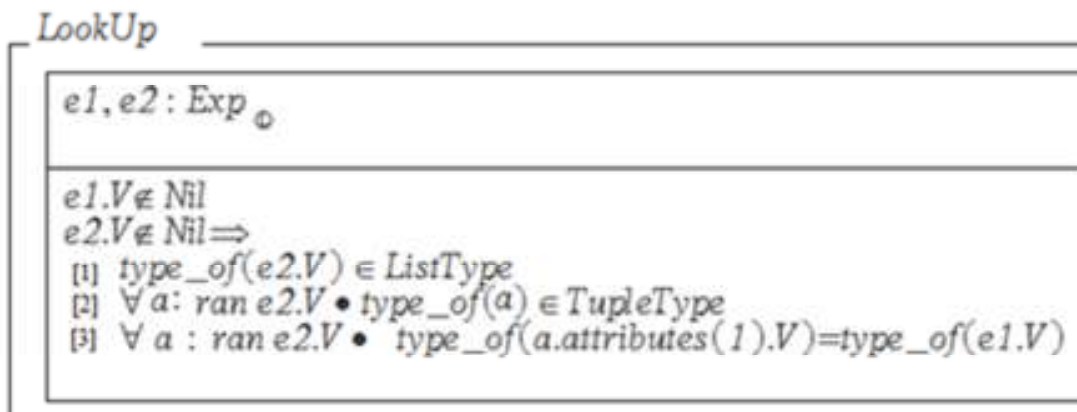
- [1] The value of **e1** should be of a list type.
- [2] The value of **e2** should be of an integer type.
- [3] The values of **e2** should be within **e1** domain (i.e. ranging from 1 to the number of element in the list **e1**) to avoid referencing errors.



5.2.3.4 LookUp Expression

The static semantics for a LookUp expression are defined as follows:

- [1] The value of **e2** should be of a list type.
- [2] The type of the elements in the list **e2** should be of a Tuple type.
- [3] The type of **e1** should be of the same type of the first element in the tuples that located in the range of the list **e2**.



5.3 Conclusion

In this chapter, we have presented formal static semantics of individual constructs in Cloud# language. These static semantics have been added to the metamodel of Cloud# language as invariants or predicates in the Object-Z class for each Cloud# syntactic construct. Some of these constructs have no static semantics so they are not presented in this chapter. On the other hand, Because of the limitations in space, we only presented the formalization of the main syntactic constructs in Cloud# language. However, the rest of the formal description is available in the appendix at the end of this thesis.

Chapter Six: A Denotational Semantics Model for Cloud#

6.1 Introduction

This chapter investigates the formalization of the denotational semantics for individual constructs in Cloud# language. The first part of this chapter presents how the Object-Z approach is used to give formal denotational semantics for Cloud# syntactic constructs. The denotational semantics will be added to the Cloud# metamodel as class operations. The last part of this chapter presents a small conclusion.

6.2 Formal Denotational Semantics Definition

The denotational semantics method aims at mapping a language construct directly to its meaning, called its denotation. The denotation is usually a mathematical function. A denotational definition is more abstract than an operational definition, for it does not specify computation steps. Its high-level and modular structure makes it especially useful to language designers and users, because the individual parts of a language can be studied without having to examine the entire definition (Schmidt, 2012).

The denotational semantics of Cloud# language are specified in terms of the operations of a hypothetical machine. The hypothetical machine is composed of: an event queue holding incoming events, an event dispatcher mechanism, and an event processor which processes dispatched event instances according to the semantics of Cloud# language. In the Object-Z approach, the denotational semantics of Cloud# syntactic constructs are captured by class operations.

This section describes the formalization of the denotational semantics for individual constructs in Cloud# language. The denotational semantics for Cloud# Computation unit with its components, Actions, and expressions will be described respectively.

6.2.1 Computation Unit (CUnit)

At run-time, a CUnit can have different states (i.e. blocked, ready, running and finished state) as shown in the figure below. Initially, any CUnit is in a ready state. When a CUnit is scheduled to execute its computation tasks, it takes the control and starts the execution. This time, its state must be changed from ready to running state.

Furthermore, there are two reasons that a CUnit returns the control and stops the execution. The first reason is because that this CUnit has already finished the execution of its computation tasks. So, its state must be changed to the finished state indicating the completion of computations. The second reason is that this CUnit is waiting for I/O event or needs to perform a privileged operation and this is usually done by Hypercalls. This time, its state must be changed to a blocked state. However, at any given time, only one CUnit is allowed to be in the running state.

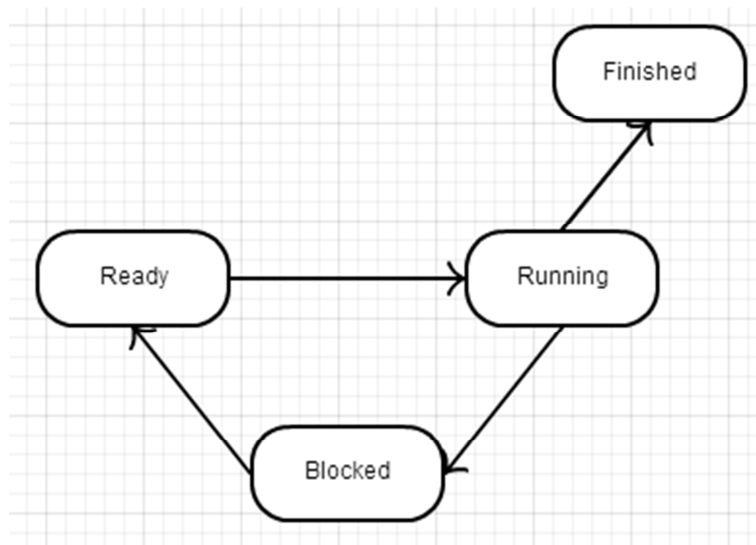
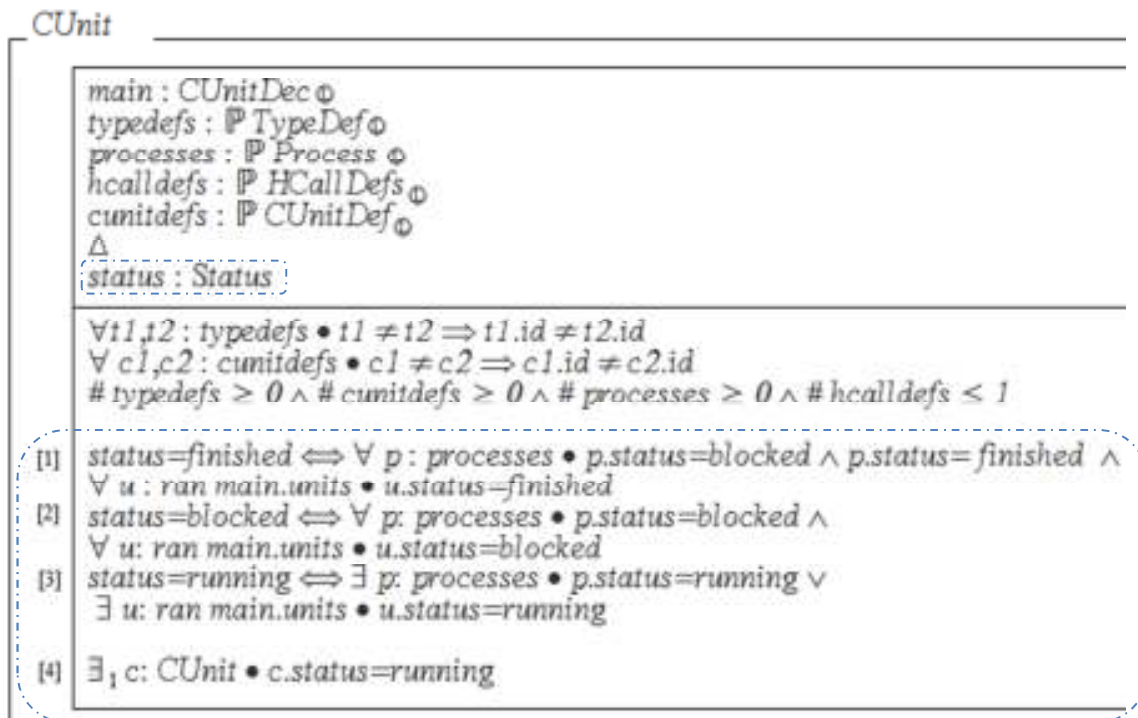


Figure 6.1: The State-transition of a Computation Unit at run-time.

The denotational semantics for a CUnit can be defined by introducing one semantic variable and five operations. The semantic variable (**Status**) is modeled as a free type in Object-Z language to indicate the different states of a given CUnit as follows:

Status ::= blocked | running | ready | finished

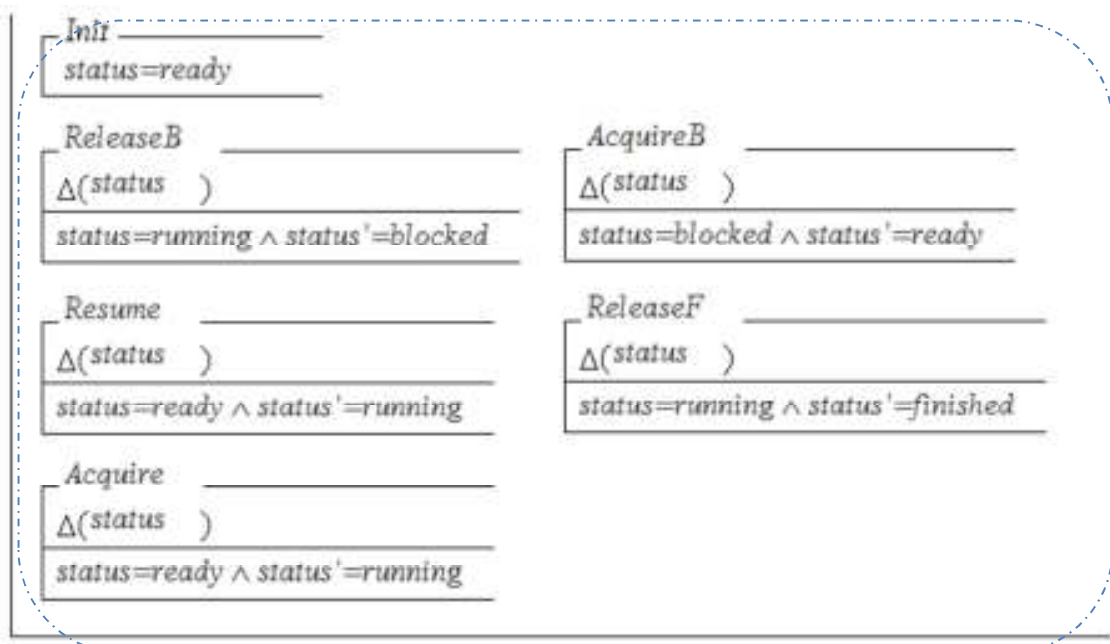
- [1] A CUnit finishes its execution and its state changes from running to finished state, if and only if all of its processes are finished and blocked. Also, all of its sub-units must be in the finished state.
- [2] A CUnit is considered to be blocked if and only if all of its processes and all of its sub-units are blocked.
- [3] A CUnit is considered to be running if and only if one of its processes or one of its sub-units is running.
- [4] The Number of running CUnits at any given time is equal to one.



The following Object-Z operations show how the state is changed for a given CUnit at run-time:

- [1] Initially, any CUnit is in a ready state.
- [2] **Acquire** operation: this is only applied when the CUnit starts its execution normally from the beginning. Its state changes normally from ready to running.

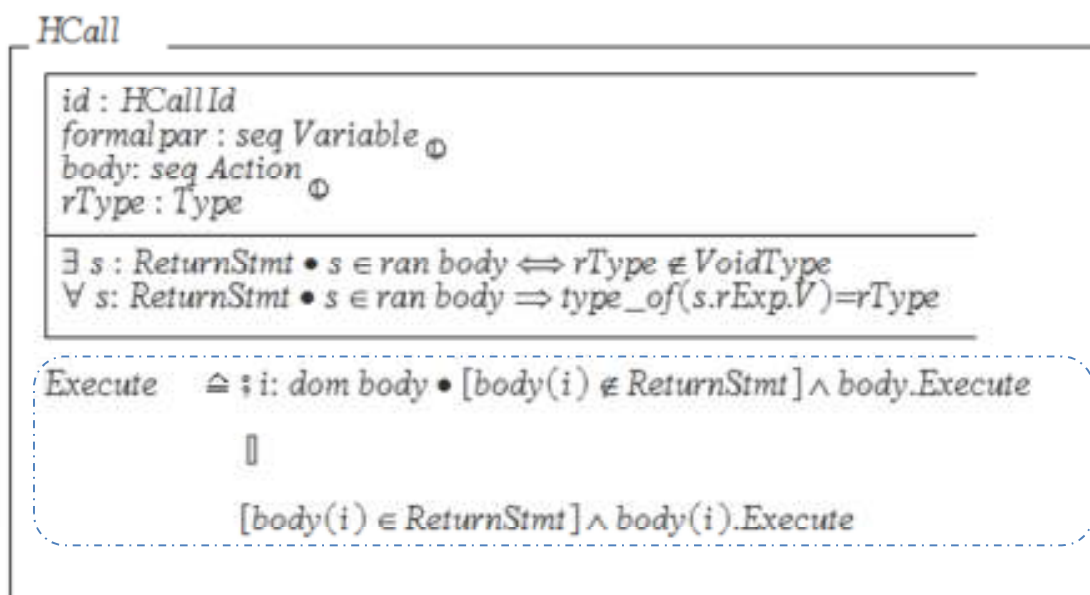
- [3] Release after finishing the execution operation (**ReleaseF**): this operation is only applied when the CUnit finishes its execution normally. Its state changes from running to finished state.
- [4] Acquire after being blocked operation (**AcquireB**): this operation is only applied when the control gets back to a CUnit after being blocked. Its state changes from blocked to ready.
- [5] Resume operation (**Resume**): when the control gets back to a CUnit after being blocked, this operation is applied so that the CUnit resumes its execution. So, its state changes from ready to running.
- [6] Release for executing a privileged operation (**ReleaseB**): this makes a CUnit to release the control to a high-privileged CUnit. Its state changes from running to blocked state.



6.2.1.1 Hypercall

A hypercall is a way for the low-privileged computation units (sub-units) to make the high-privileged computation unit (**CUnit**) to handle privileged operations. The denotational semantics for a hypercall are defined simply by executing the body of the HCall as follows:

- [1] **Execute** operation: it executes the actions in the body of an HCall sequentially. However, if some of these actions belong to a Return statement, it executes this action and terminates the execution of that HCall.



6.2.1.2 Interleaving Concurrency

There are two different kinds of concurrency: non-interleaving and interleaving concurrency. The non-interleaving concurrency assumes that the execution of an action is non-interruptible. So the modeling of non-interleaving concurrency is very simple and requires no new concepts. On the other hand, modeling interleaving concurrency is more complicated. For instance, when two assignment-statements evaluate concurrently on the same store, the result is a set of possible output stores as shown in the example below.

Example: Consider $(X := X+2 ; X := X-1) \parallel (X := 3)$, where \parallel is a parallel operator with interleaving concurrency. Even if we consider that each assignment is atomic, there will be a set of possible interleaving of the statements that lead to different output store as follows.

$X:=X+2; X:=X-1; X:=3$
 $X:=X+2; X:=3; X:=X-1$
 $X:=3; X:=X+2; X:=X-1$

This means that we need a form of denotational semantics for representing the operational aspects of concurrency in the semantics. In this thesis we follow the resumption denotational semantics that presented in (Schmidt, 1997). The resumption semantics approach assumes that each action or command is a sequence of steps called **Evaluation Steps**. Each action or command in a language has different evaluation steps, For example, the evaluation steps for the simple addition $\cdot X + Y \cdot$ involves the following steps:

- ✓ Evaluate the left operand.
- ✓ Evaluate the right operand.
- ✓ Add.

Consider $A = \cdot a_1, a_2, \dots, a_n \cdot$, $B = \cdot b_1, b_2, \dots, b_n \cdot$, and $P = A; B$, where A and B are two different actions within the process P . The action A is supposed to be executed before the action B . In interleaving concurrency, there are two conditions that must hold for legal execution of a process:

1. The sequence of actions within the same process must be preserved. That is, the action A must be executed before the action B .
2. The sequence of evaluation steps with a single action must be preserved. That is, in the action A , for instance, the execution of its evaluation steps proceeds as follows: $a_1; a_2; a_3 \dots$ etc.

In this section we show how the interleaving concurrency can be applied to Cloud# language.

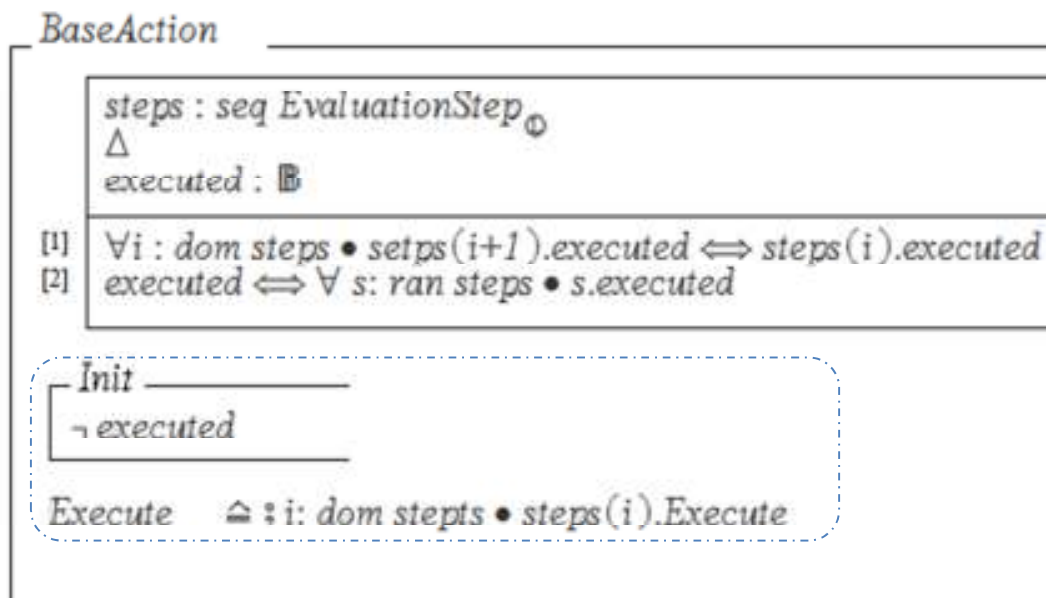
✓ Cloud# Action with Interleaving Concurrency

As mentioned above, every action is composed of a sequence of evaluation steps. So, we have introduced a new class named **BaseAction** in which all the actions in Cloud# language are inherited from. This class has one attribute named **steps** which indicates a sequence of evaluation steps for an action. However, the formalization of the class

EvaluationStep is not mentioned in this thesis, because we assume that the number of evaluation steps for a specific action depends on the machine that executes such action.

The denotational semantics for the class **BaseAction** are defined by introducing one semantic variable (**executed**) and one operation (**Execute**) as follows:

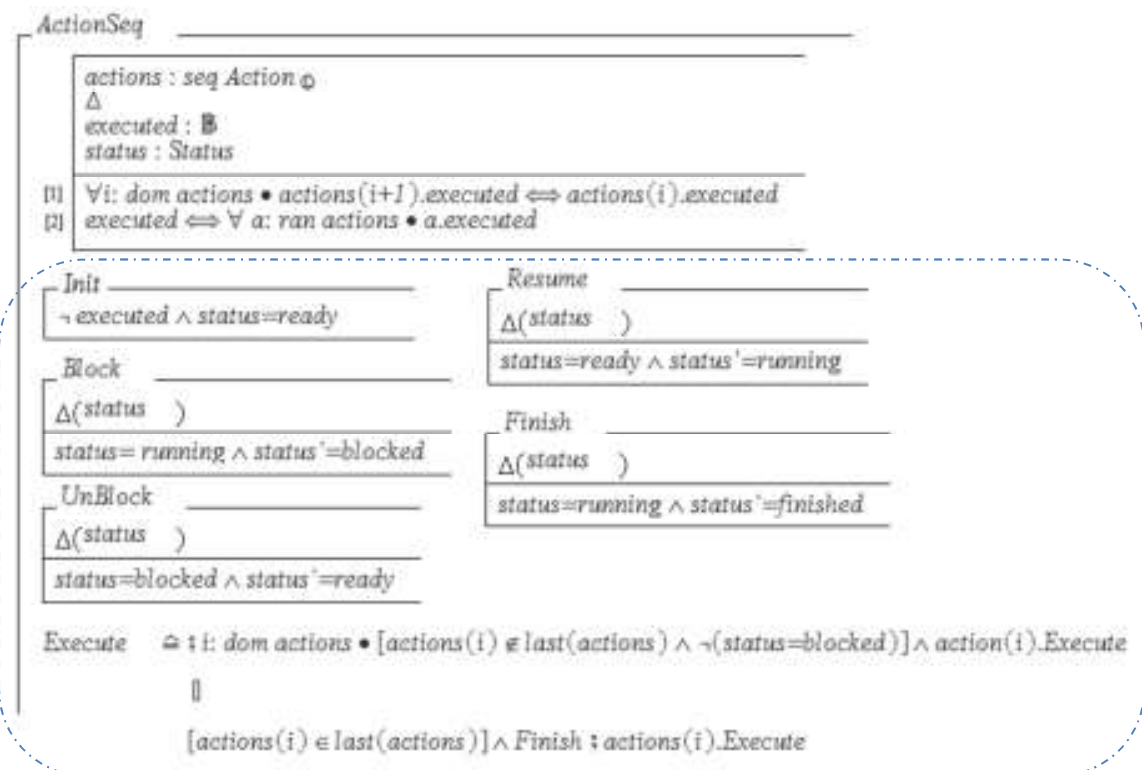
- [1] An evaluation step can be executed, if and only if its precedent evaluation step in the same action is executed.
- [2] An action is considered to be executed, if and only if all of its evaluation steps have been executed.
- [3] The Execute operation executes the evaluation steps for a specific action in a sequential manner.



✓ Action Sequence

The denotational semantics for an action sequence in Cloud# can be defined by introducing two semantics variables and five operations. The semantic variable (**executed**) refers to the completion of execution for an action sequence, while the semantics variable (**status**) refers to the different states (i.e. blocked, running, etc.) of an action sequence at run-time. The operations show how the state for an action sequence is changed as follows:

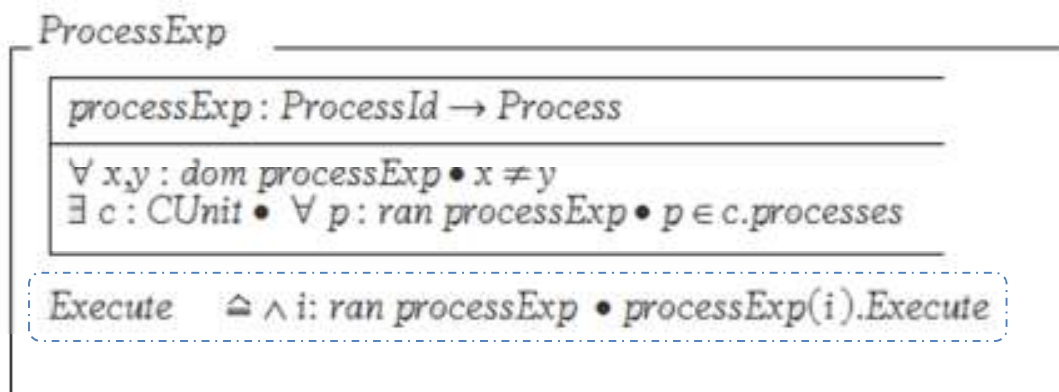
- [1] An action can be executed, if and only if its precedent action in the same action sequence is executed.
- [2] An action sequence is considered to be executed, if and only if all of its actions have been executed.
- [3] Initially, an action sequence is in a ready state and not executed.
- [4] The **Block** operation blocks the execution of an action sequence and changes its state to blocked state.
- [5] The **UnBlock** operation changes the state of an action sequence to a ready state and makes it ready for execution.
- [6] The **Resume** operation changes the state of an action sequence to the running state and makes it start execution.
- [7] The **Execute** operation executes the actions within an action sequence in a sequential manner as the state of an action sequence is not blocked. On the other hand, if the action sequence reaches to the last action, its state is changed to the finished state.



✓ Process Expression

The denotational semantics for a process expression are defined by one operation (**Execute**) as follows:

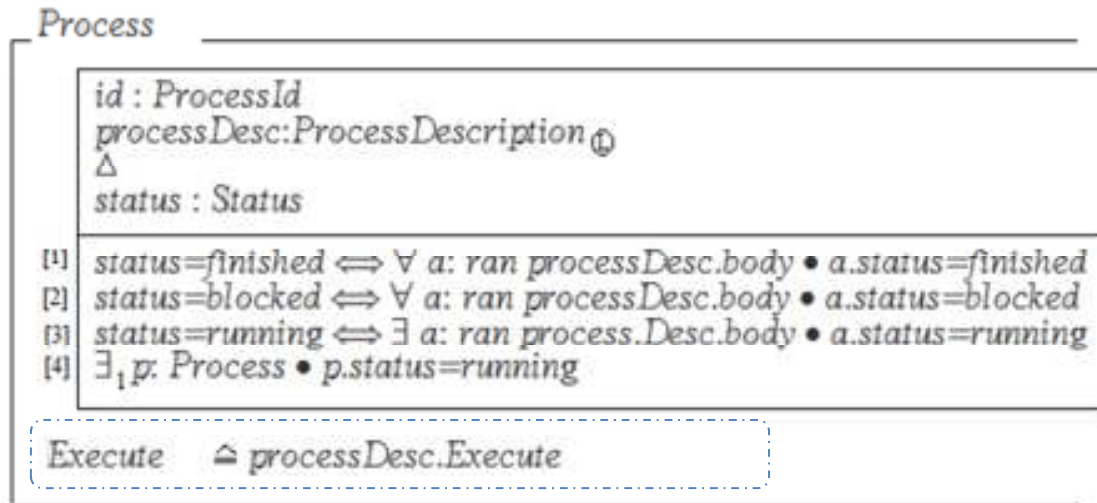
[1] The Execute-operation executes all processes that are referred in the ProcessExp concurrently.



✓ Process

The denotational semantics for a Process are defined by introducing one semantic variable (**status**) and one operation (**Execute**). The semantic variable refers to the different states (i.e. running, blocked, etc) of a process at run-time as follows:

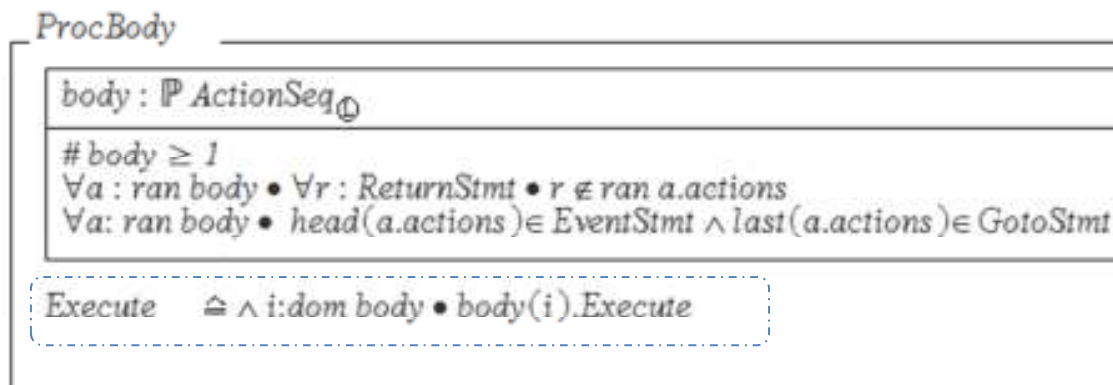
- [1] A process is considered to be finished, when all of its action sequences are finished.
- [2] A process is considered to be blocked, when all of its action sequences are blocked.
- [3] A process is considered to be running, if one of its action sequencing is running.
- [4] At any given time, only one process is allowed to be in the running state.
- [5] The Execute-operation executes the body of a process.



✓ Process Body

The denotational semantics for a process body are defined by one operation (**Execute**) as follows:

[1] The Execute-operation executes all action sequences that present in a process body concurrently.



6.2.2 Cloud# Actions

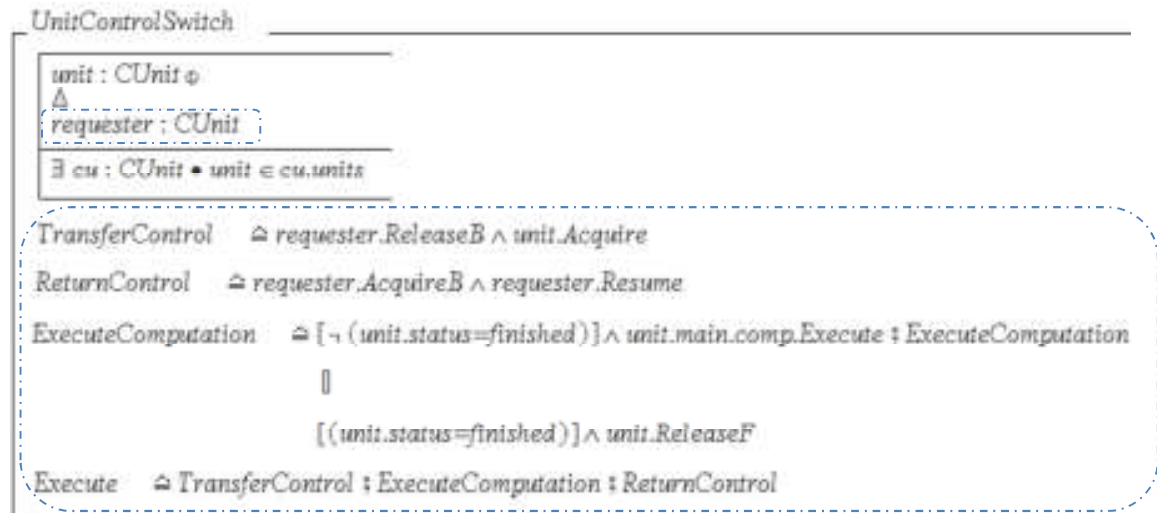
This section investigates the formalization of the denotational semantics for Cloud# actions. The meaning of an action can be modeled as a store transformation which is captured in class operations.

6.2.2.1 Unit Control-Switch Statement (· CUnit ·)

Cloud# language provides two directions of control and data transfer (i.e. from the high-privileged computation units to the low-privileged computation units and vice versa). If the high-privileged computation unit wishes to run a low-privileged computation unit, the unit control-switch statement is used. That is, the high-privileged computation unit releases the control to the low-privileged computation unit which takes the control and starts execution of its computation tasks. The low-privileged computation unit only returns the control if it has already finished the execution of its computation tasks or if it is blocked (i.e. needs to perform a privileged operation and this is usually done with Hypercalls).

In the dynamic semantics of the Unit Control-Switch statement, the control can be returned back to the high-privileged computation unit only and only if the low-privileged computation unit finishes the execution of its computation tasks. We do not care about returning the control when the low-privileged computation unit is blocked, because this can be handled in the Hypercall statement. The denotational semantics for Unit Control-Switch action can be described by introducing one semantic variable and four operations. The semantic variable refers to a high-privileged CUnit that are executing the Unit Control-Switch statement. The Object-Z operations are described as follows:

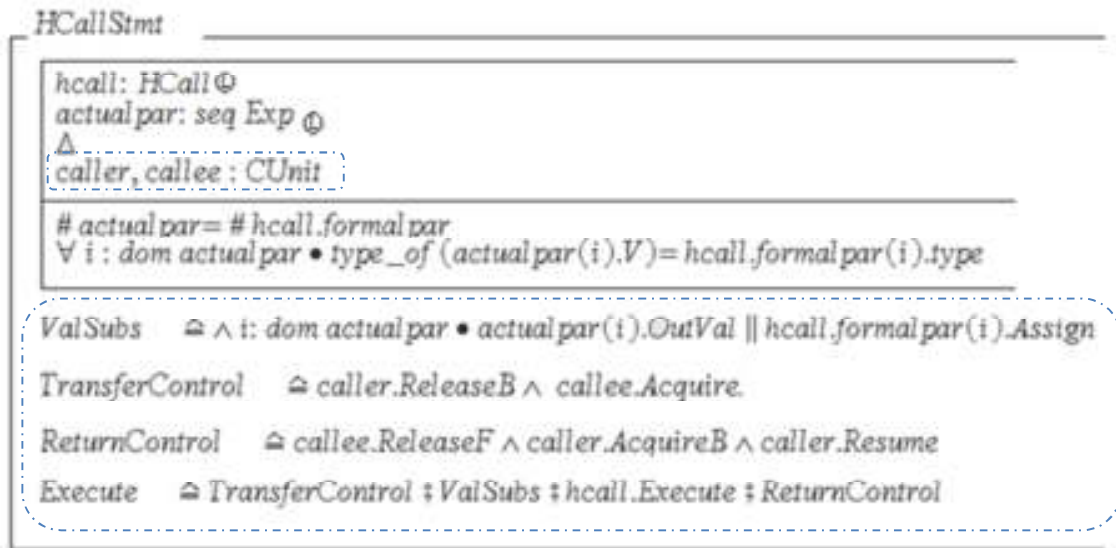
- [1] In the **TransferControl** operation, the requester which is a high-privileged CUnit blocks itself and releases the control. On the other hand, the requested low-privileged CUnit acquires the control and starts the execution of its computation tasks from the beginning.
- [2] In the **ReturnControl** operation, the requested CUnit releases the control after finishing its computation tasks and the requester CUnit acquires the control after being blocked and resumes its execution.
- [3] In the **ExecuteComputation**, the requested CUnit keeps executing its computation tasks until the execution is finished and then it releases the control back to the high-privileged CUnit.
- [4] The **Execute** operation is used to preserve the sequence of operations.



6.2.2.2 HCall Statement

HCall statement is used by a low-privileged CUnit for asking the high-privileged one to handle a privileged operation. The denotational semantics for an HCall statement are defined by introducing two semantics variables (**caller** and **callee**) and four operations. The semantics variables refer to two different Computation Units. The caller refers to the computation unit that wants to perform a privileged operation, while a callee refers to a high-privileged computation unit that supposed to perform such operation. The Object-Z operations are described as follows:

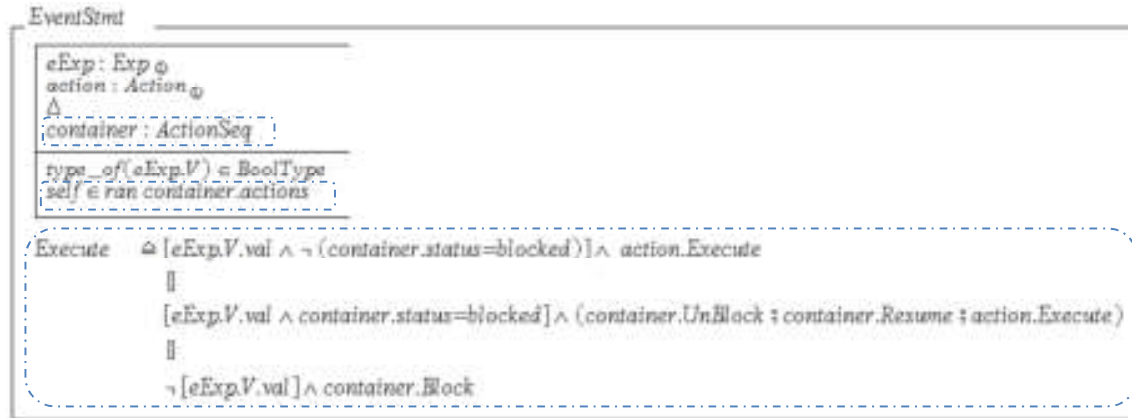
- [1] **TransferControl** operation: The caller blocks itself and transfer the control to the callee.
- [2] **ValSubs** operation: the values of the parameters in the HCall statement are assigned to the formal parameters in the HCall.
- [3] **ReturnControl** operation: the callee releases the control after executing the HCall. This time, the caller acquires the control again and resumes its execution.
- [4] **Execute** operation: it is used for executing the HCall and sequencing the operations.



6.2.2.3 Event Statement ($E \Rightarrow A$)

The Event Statement ($E \Rightarrow A$) means that if the expression **E** evaluates to true, then the action **A** proceeded to execute; otherwise, the action is blocked until E becomes true. Event statement is useful for modeling event-driven systems. For instance, event statement is used to model a network driver, which responds only to the arrival of new network packets. The denotational semantics for an Event Statement can be defined by introducing one semantic variable (**container**) and one operation (**Execute**). The semantic variable refers to the container in which the event statement is located. The operation is described as follows.

- [1] **Execute** operation: if the value of the expression (**eExp**) evaluates to true and the container is not blocked, then the action proceeds to execute; otherwise the container blocks itself and waits for an event. Once the value of **eExp** becomes true, the container unblocks itself, resumes its execution and executes the action.

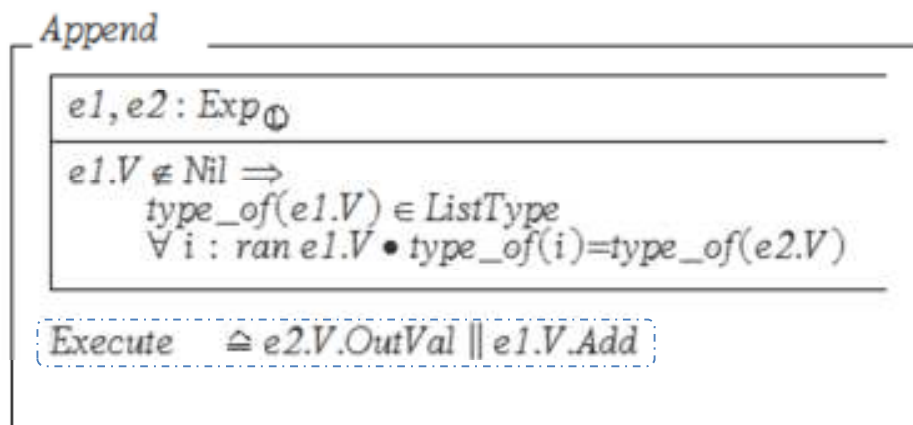


6.2.2.4 Append Action

Append (**e1**, **e2**) is used to add the element (**e2**) on the top of the list (**e1**). The denotational semantics for an Append Action are defined by introducing one operation (**Execute**) as follows:

[1] The Execute-operation adds the value of the expression (**e2**) to the top of the list (**e1**).

Please refer to the appendix in the end of this thesis to see how the value is added or removed from a list value.

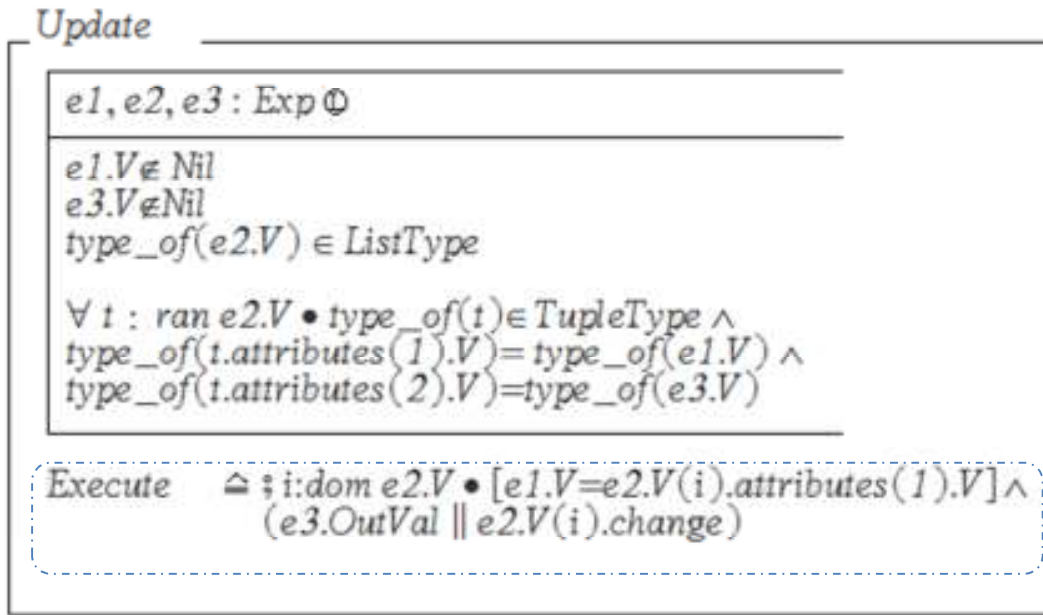


6.2.2.5 Update Action

Let **e1**, **e3** are expressions and **e2** is a list that contains elements of a tuple type. **Update** (**e1**, **e2**, **e3**) is used to change the second element of a tuple to **e3**, if the first element of

that tuple equals to $e1$. The denotational semantics for an Update Action are defined by introducing one operation (**Execute**) as follows:

[1] The Execute-operation changes the value of the second element in a Tuple to $e3$, if the value of the first element in that Tuple equals to the value of the expression ($e1$).



Please refer to the appendix in the end of this thesis to see how the value of a Tuple is changed.

6.2.2.6 noop Action

The noop action does nothing when executed and is usually used in the computation task component when it has nothing to do. The denotational semantics for the noop action are defined by introducing one operation (**Execute**) as follows:

[1] The Execute-operation does nothing when executed.



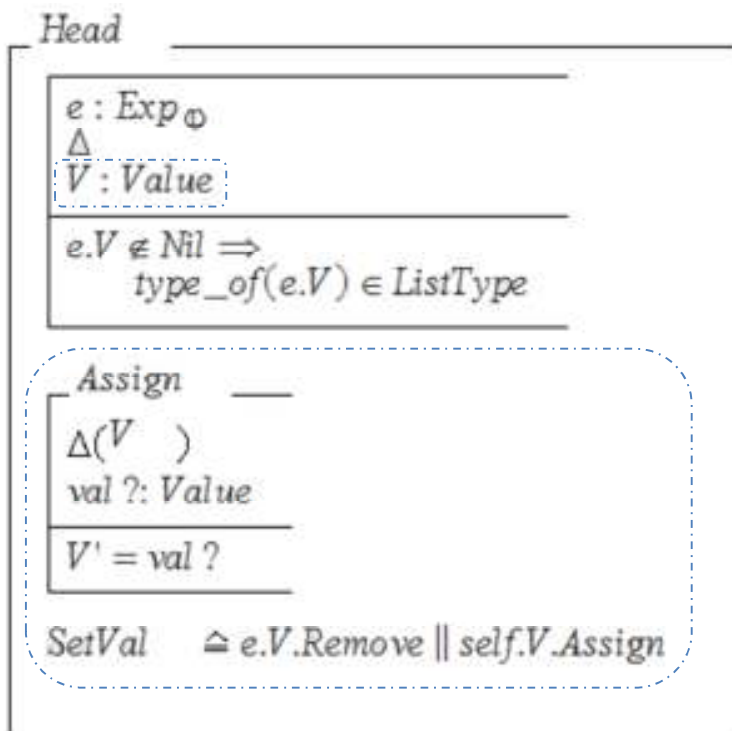
6.2.3 Cloud# Expressions

This section presents the denotational semantics for Cloud# Expressions. The denotational semantics for an expression are defined by the final value of that expression.

6.2.3.1 Head Expression

Head expression evaluates to the value of the first element in a list and removes the first element from that list. The denotational semantics for the Head expression are specified by introducing one variable (**V**) and two operations as follows:

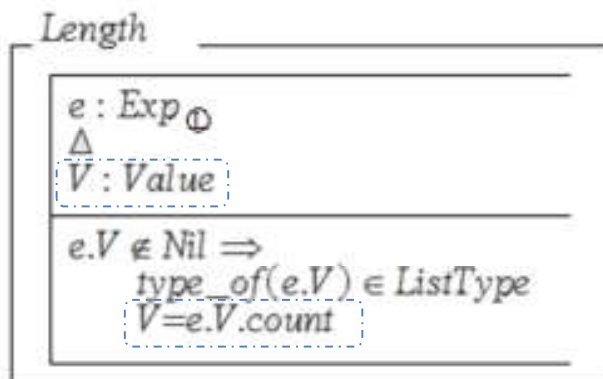
- [1] The variable (**V**) is equal to the value of the first element in the list.
- [2] The **Assign** operation changes the value (**V**) of the Head Expression.
- [3] The **SetVal** operation assigns the value of the first element in the list (**e**) to the variable (**V**) and removes that element from the list.



6.2.3.2 Length Expression

Length expression evaluates to the number of elements in a list. The denotational semantics for Length expression are defined by introducing one variable (**V**) as follows:

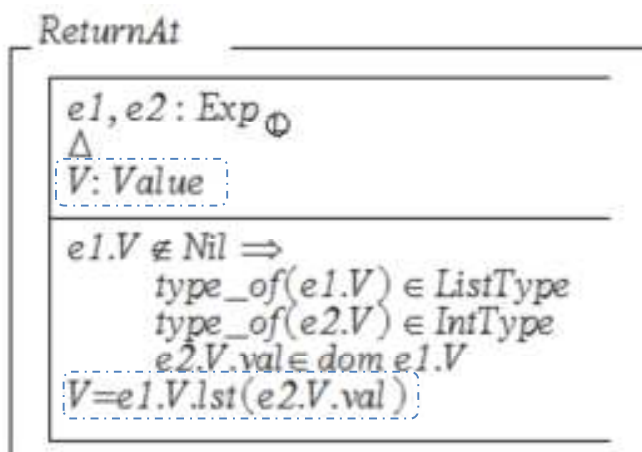
- [1] The variable (**V**) equals to the number of elements in the list (**e**).



6.2.3.3 ReturnAt expression (E[i])

ReturnAt expression evaluates to the value of the i^{th} element in a list. The denotational semantics for **ReturnAt** expression are specified by introducing one variable (**V**) as follows:

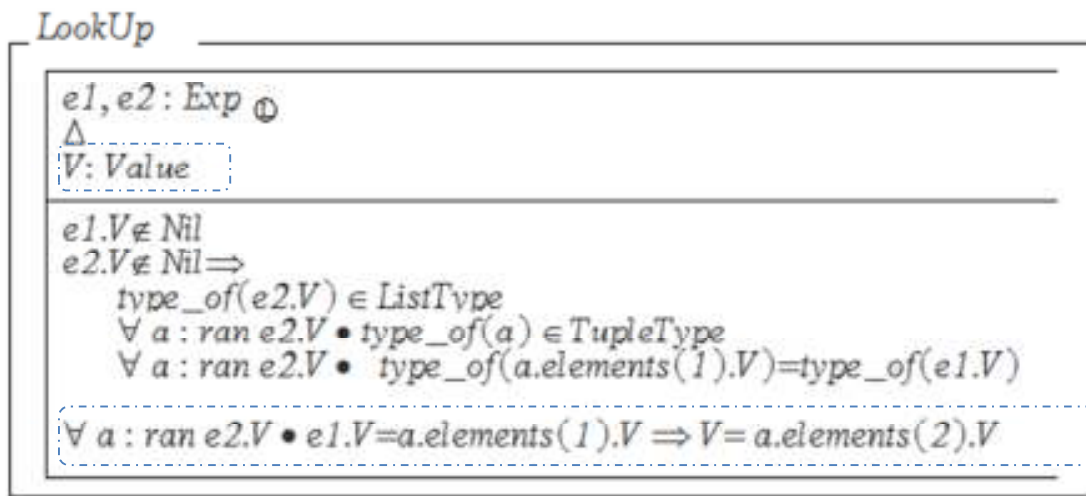
- [1] The variable (**V**) equals to the value of the element in the list (**e1**) which is indexed by the value of **e2**.



6.2.3.4 LookUp Expression

Let $e1$ is an expression and $e2$ is a list that contains elements of a tuple type. **LookUp** ($e1, e2$) expression evaluates to the second element in a tuple, if the first element of that tuple equals to $e1$. The denotational semantics of LookUp expression are specified by introducing one variable (V) as follows:

- [1] The variable (V) equals to the value of the second element in a Tuple, if the value of $e1$ is equal to the value of the first element in that Tuple.



6.3 Conclusion

In this chapter, we have presented formal denotational semantics for individual constructs in Cloud# language. These semantics have been added to the metamodel of Cloud# language as class operations. Because of the limitations in space, we only presented the formalization of the main syntactic constructs in Cloud# language. However, the rest of the formal description is available in the appendix at the end of this thesis.

Chapter Seven: Evaluation

7.1 Introduction

This chapter presents an evaluation for the work that has been done in this thesis. It investigates the consistency of Cloud# language. The first part presents a background on the usual techniques for verification and validation that can be done depending on the denotational semantics of a language. The second part presents a case study which illustrates how to convert Cloud# models to OZ specifications. The consistency of Cloud# language is discussed in the last part of this chapter.

7.2 Background on the Formal Verification and Validation Techniques

Formal methods provide precise ways to specify modeling languages using formal notations with a well-defined syntax and precise semantics. These notations have a set of associated tools and inference rules which enable automated analysis, verification and validation (Coppit, 2003).

There are two main techniques for verification and validation that can be done depending on the denotational semantics of a language: the consistency checking and proving interesting properties about language models. This section investigates each technique according to its usage and how it is handled.

7.2.1 Checking the consistency of a language

One of the most important issues that must be proved about any language is its consistency (Coppit, 2003). The consistency checking for any language mainly involves two steps. The first step is to ensure that the formal semantics definition truly expresses the right semantics for the language. This type of consistency checking requires working with the language designers, because they are the only ones who know the actual meaning of the available informal semantics definition. The second step is to ensure the soundness of the language by checking the agreement between the formal semantics definition and the available concrete syntax (i.e. type-checking). This is usually done using the current existing tools for formal validation (i.e. Z/EVES, CZT, etc). For instance, the authors in (Wang et al, 2012) applied an Object-Z type checker to check the

consistency of the WSMO language. They used Amazon Associates Web service model (A2S) as a case study. Amazon model was first transformed to Object-Z language and it was loaded with the formal semantics definition to an Object-Z type-checker to ensure if the language sound. Amazon has been chosen because it is one of the largest models that have been developed based on this language. In other words, a selected model must be large enough so that it can cover the most aspects of the language, otherwise the checking results won't be accurate. In this thesis, we use the same approach that is used for WSMO to check the consistency of Cloud# language.

7.2.2 Proving some important properties about language models

The second step after checking the consistency of a language is to reason about some interesting properties that should be exist in the language models (i.e. liveness properties, safety properties, etc.). This is usually done using mathematical proof techniques (i.e. based on temporal logic). It mainly involves two steps: the first step is to state or write the desired property in an informal way (i.e. natural language) and then it is translated to a logical expression. Once the property is converted to a logical expression, the second step starts by applying one of the mathematical proof techniques (i.e. proof by induction) to check if such property holds or not. For instance, the authors in (Taguchi and Ciobanu, 2004) used proof by induction to prove the liveness properties for concurrent Z specifications.

7.3 A Case Study—Cloud# Basic Model

In this section, we use the Cloud# basic model that appears in (Liu and Zic, 2011) as a case study to illustrate how it can be represented in Object-Z specifications. This model shows a cloud service which allows multiple virtual machines to run and share their storages and network resources. The features showed by this model are: a basic scheduler, the isolation of storage space for different clients and the virtualization of network devices. The basic Cloud# model is represented in Object-Z specifications depending on the formal semantics definition that we have defined in the previous chapters. That is, any concrete Cloud# element is modeled as an instance of the Object-Z class that represents its abstract syntax, static and dynamic semantics. Please refer to

(Smith, 1992) for more information about object instantiation in Object-Z language. Because of the limitations in space, we will only present the main parts of this model.

7.3.1 The Part of Computation Units

The main Computation Unit in the Cloud# basic model includes two virtual machines (VM1 and VM2) as shown below. The computation task in this model is a parallel composition of Scheduler and Network which performs scheduling and packet processing.

```
(vmm, Scheduler | Network, [VM1, VM2], HCallDefs, Conf, Res)
VM1 = (vm1, noop, [], {}, {}, VmRes)
VM2 = (vm2, noop, [], {}, {}, VmRes)
```

This can be modeled in OZ as follows:

```
cunit, VM1, VM2 : CUnit
-----
cunit.main = dec
cunit.typedefs = {loc, val, src, dest, id, vloc, ploc, ip, ready, runvmid}
cunit.processes = {Scheduler, Network}
cunit.hcalldefs = {hcalldef}
cunit.cunitdefs = {VM1, VM2}
VM1.main = dec1
VM1.typedefs = {src1, dest1, val1}
VM1.processes = {}
VM1.hcalldefs = {}
VM1.cunitdefs = {}
VM2.main = dec2
VM2.typedefs = {src2, dest2, val2}
VM2.processes = {}
VM2.hcalldefs = {}
VM2.cunitdefs = {}

dec, dec1, dec2 : CUnitDec
-----
dec.id = vmm
dec.comp = {(scheduler, Scheduler), (network, Network)}
dec.units = {VM1, VM2}
dec.hcalls = hcalldef
dec.configuration = rconf
dec.resource = rres
dec1.id = vm1
dec1.comp = noop
dec1.units = {}
.
.
.
dec2.id = vm2
dec2.comp = noop
dec2.units = {}
.
.
.
```

7.3.2 The Part of Processes

The Cloud# basic model includes two main processes (Scheduler and Network) as shown below. The Scheduler is represented as a set of actions that are executing in a sequential manner. On the other hand, the Network process is a composite process. It is represented as a parallel composition of two different processes (Send and Receive).

```
Scheduler=starts:
  for vm in self.units do
    rdy:=lookup(vm.id,self.conf.status);
    if rdy=1 then
      self.conf.cur:=vm.id;
      [vm];
      update(vm.id,self.conf.status,0);
    goto starts;
Network=Receive|Send
.
.
.
```

This can be modeled in OZ as follows:

```
Scheduler, Network, Send, Receive : Process
```

```
Scheduler.id= scheduler
Scheduler.processDesc = pbody
Network.id=network
Send.id= send
Send.processDesc= sendbody
Receive.id= receive
Receive.processDesc=receivebody
Network.processDesc= ((send, Send), (receive, Receive))
```

```
pbody, sendbody, receivebody : ProcBody
```

```
pbody.body= {starts}
sendbody.body= {startd1, startd2}
receivebody.body= {starttr}
```

```
starts, startd1, startd2, starttr : ActionSeq
```

```
starts.actions= (label, for, goto)
startd1.actions= (event1, assign1, append1, goto1)
startd2.actions= (even2, assign2, append2, goto2)
starttr.actions = (rEvent, rAssign1, rAssign2, rFor, rGoto)
```

7.3.3 The Part of Hypercalls

In the Cloud# basic model, there is a definition for two different Hypercalls (read and write) as shown below. These calls enable the virtual machines to access the physical storages, retrieve and update the data in the cloud infrastructure.

```

HCallDefs={
  read(loc) =
    mmu:=lookup(self.conf.cur,self.conf.sm);
    ploc:=lookup(loc,mmu);
    x:=lookup(ploc,self.res.storage);
    return x;
  write(loc,E) =
    mmu:=lookup(self.conf.cur,self.conf.sm);
    ploc:=lookup(loc,mmu);
    update(ploc,self.res.storage,E);
}

```

This can be modeled in OZ as follows:

<i>hcalldef, hcalldef1, hcalldef2 : HCallDefs</i>

<i>hcalldef.hcalls = {read, write}</i>
<i>hcalldef1.hcalls = { }</i>
<i>hcalldef2.hcalls = { }</i>

<i>read, write : HCall</i>

<i>read.id=read</i>
<i>read.formalpar={loc }</i>
<i>read.body={ hAssign1, hAssign2, hAssign3, hReturn}</i>
<i>read.rType= val</i>
<i>write.id=write</i>
<i>write.formalpar={loc, E }</i>
<i>write.body={wAssign1, wAssign2, wUpdate}</i>
<i>write.rType=void</i>

7.4 Conclusion

We used the Object-Z version of the Cloud# basic model that has been discussed in the previous section to check the consistency of Cloud# language. The basic Cloud# model has been applied along with the existing formal denotational semantics definition of Cloud# language to the Zed Community Tool (CZT) type-checker. No typing errors have been found which indicates the consistency of Cloud# language. However, we can't state precisely that Cloud# language is consistent, since the selected model is not large enough to cover the most aspects of the language. However, this model has been chosen due to the fact that Cloud# is still new and has not been used to model real and large cloud infrastructures. Well, a more complex and large model leads to a more accurate result. Now that this language has been provided with a formal denotational semantics definition, it is easy to use this language for modeling real and large Cloud infrastructures and check the consistency of Cloud# language in a more accurate way.

Chapter Eight: Conclusions and Future Works

8.1 Conclusions

In this thesis, we have presented a formal denotational semantics for Cloud# language which is a domain-specific modeling language for modeling the infrastructure of the Cloud. Object-Z language has been used as a meta-language for defining the formal semantics of Cloud#. The formal semantics definition with Object-Z language has been given as a single unified framework. That is, the abstract syntax, static and dynamic semantics of a single language construct is specified in one Object-Z class. Not only does this help the readability of the semantics, but if the language is enhanced or evolved, the required modifications can be done by the minimal disruption to the existing semantics. Also it is possible to use some parts of semantics definition of one language to define another.

On the other hand, the consistency checking for Cloud# language has been done using an Object-Z type-checker tool. A sample Cloud# model has been converted to the Object-Z specifications and then applied along with the existing formal denotational semantics to a type-checker. No typing errors have been found which indicates the consistency of Cloud# language. However, it is not realistic to state precisely that Cloud# language is consistent, because the sample model is not large enough to cover the most aspects of the language and it doesn't even represent a real cloud infrastructure. This model has been chosen due to the fact that Cloud# is still new and has not been used to model real and large cloud infrastructures. Well, a more complex and large model leads to a more accurate result.

8.2 Future works

Defining the formal semantics of a modeling language can be beneficial to that language in many different ways (i.e. for reasoning about the language or for providing tool support). In this thesis, we have defined a formal denotational semantics for Cloud# language. This type of semantics facilitates reasoning about the language using the existing tools for verification and validation or by using mathematical proof techniques. In this thesis, we only used one of the existing tools to check the consistency of Cloud# language. However, in future, some mathematical proof techniques will be adopted to reason about Cloud# models and to check the existence of some interesting properties such as the liveness property as mentioned in chapter 7.

On the other hand, the denotational semantics definition is not enough. Cloud# is like any other language, it also needs an operational semantics definition. The operational semantics definition will help in developing tools (i.e. Compiler, interpreter, etc) for Cloud# language. Without the operational semantics definition, the generation of these tools will be a very complex process. In our future work, we propose to provide Cloud# language with an operational semantics definition which is consistent to the already existing denotational semantics. This can be accomplished by using one of Object-Z language extensions with a process-based language (i.e. pi-calculus, CSP, etc). That is, all the aspects of the language will be defined in a single unified framework. Similar work for defining all the aspects of the language in a single framework appears in (Hahn, 2008).

REFERENCES

- Adamis, G., Horváth, R., Pap, Z., and Tarnay, K. (2005). Standardized languages for telecommunication systems. *Computer Standards & Interfaces*, 27(3), 191-205. doi:10.1016/j.csi.2004.09.005.
- Andova, S., Van den Brand, M., and Engelen, L. (2011). Prototyping the Semantics of a DSL using ASF+SDF: Link to Formal Verification of DSL Models. *Electronic Proceedings in Theoretical Computer Science*, 56, 65–79. doi:10.4204/EPTCS.56.5
- Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., and Neugebauer, R. (2003). Xen and the Art of Virtualization Categories and Subject Descriptors. *Computer Science and Information System*.
- Bruneli, H. (2010). Combining Model-Driven Engineering and Cloud Computing. modeling design and analysis for the Service Cloud- MDA4ServiceCloud'10: Workshop's 4th edition. Paris, France. ECMFA.
- Bryant, B., Gray, J., Mernik, M., Clarke, P., France, R., and Karsai, G. (2011). Challenges and directions in formalizing the semantics of modeling languages. *Computer Science and Information Systems*, 8(2), 225–253. doi:10.2298/CSIS110114012B
- Chen, K., Sztipanovits, J., and Abdelwalhed, S. (2005). Semantic anchoring with model transformations. *Model Driven*, 115–129. Springer. Verlag. Retrieved from <http://www.springerlink.com/index/y3g6237844251u27.pdf>
- Cho, H., Sun, Y., and Gray, J. (2011). Key Challenges for Modeling Language Creation By Demonstration. *2011 Workshop on Flexible Modeling*. Retrieved from http://www.ics.uci.edu/~nlopezgi/flexitoolsICSE2011/papers/cho_flexitools_icse2011.pdf.
- Coppit, D. (2003). Engineering modeling and analysis: Sound methods and effective tools. Syntax, (January). Computer Science PHD Thesis. University of Virginia.

- Dong, J, and Duke, R. (1993). Class union and polymorphism. *Technology of Object-Oriented Language* Retrieved from <http://www.comp.nus.edu.sg/~dongjs/papers/tools93.ps>
- Dong, J. S. (2005). An Object Semantic Model of SOFL, In *Intergrated Formal Methods (IFM'99)*, Pages 189-208, York, UK,(10139236).
- Dong, JS, Duke, R., and Rose, G. (1997). An object-oriented denotational semantics of a small programming language. *Object Oriented Systems*, 4(1), 29–52. Retrieved from <http://www.comp.nus.edu.sg/~dongjs/papers/oos97.ps>
- Engelen, L., and Van den Brand, M. (2010). Integrating Textual and Graphical Modelling Languages. *Electronic Notes in Theoretical Computer Science*, 253(7), 105–120. doi:10.1016/j.entcs.2010.08.035
- Engine, C. (2010). domain-specific languages Domain-Specific Languages in a Customs. *Information System*, (April), 65-71.
- Fisher, D. (2010). Static semantics for syntax objects.PHD thesis, Computer and Information Science.Univeristy of notheasten Boston.
- Gargantini, A., Riccobene, E., and Scandurra, P. (2009). A semantic framework for metamodel-based languages. *Automated Software Engineering*, 16(3-4), 415–454. doi:10.1007/s10515-009-0053-0
- Goncalves, G., Endo, P., Santos, M., Sadok, D., Kelner, J., Melander, B., and Mangs, J.-E. (2011). CloudML: An Integrated Language for Resource, Service and Request Description for D-Clouds. *2011 IEEE Third International Conference on Cloud Computing Technology and Science*, 399–406. doi:10.1109/CloudCom.2011.60.
- Griffiths. A, and Rose.G (1995), A Semantic Foundation for Object Identity in Formal Specification, *Object-Oriented Systems*, vol. 2, Chapman & Hall, pp. 195–215.

- Hahn, C. (2008). A domain specific modeling language for multiagent systems. *on Autonomous agents and multiagent systems*, (Aamas), 233–240. Retrieved from <http://dl.acm.org/citation.cfm?id=1402420>
- Ji, X. (2011). Analysis and Design for Object-oriented Multi-tier Architecture of Public Opinion Survey System Based on UML. *Procedia Engineering*, 15, 5445–5449. doi:10.1016/j.proeng.2011.08.1010
- Kiel, C., and Schneider, C. (2011). On Integrating Graphical and Textual Modeling. Diploma Thesis. University of KIEL.
- Kos, T., Kosar, T., and Mernik, M. (2011). Development of data acquisition systems by using a domain-specific modeling language. *Computers in Industry*. doi:10.1016/j.compind.2011.09.004.
- Lara, J. De, Vangheluwe, H., and Alfonseca, M. (2004). Meta-modelling and graph grammars for multi-paradigm modelling in AToM 3. *Software and Systems Modeling*. Retrieved from <http://www.springerlink.com/index/atmw5mtfqa33cmh1.pdf>
- Lester, D. (2011). Module COMP36411 Understanding Programming Languages. *Electronic Notes in Theoretical Computer Science* Univeristy of Manchester.
- Liu, D., and Zic, J. (2011). Cloud#: A Specification Language for Modeling Cloud. *2011 IEEE 4th International Conference on Cloud Computing*, 533–540. doi:10.1109/CLOUD.2011.18
- Ma, Z. M., Yan, L., and Zhang, F. (2012). Modeling fuzzy information in UML class diagrams and object-oriented database models. *Fuzzy Sets and Systems*, 186(1), 26–46. doi:10.1016/j.fss.2011.06.015
- Naumenko, A., Wegmann, A., and Atkinson, C. (2003). The Role of Tarski's Declarative Semantics in the Design of Modeling Languages. *Swiss Federal Institute*. Retrieved from http://ic2.epfl.ch/publications/documents/IC_TECH_REPORT_200343.pdf

- Pichler, P., Weber, B., and Zugal, S. (2012). Imperative versus Declarative Process Modeling Languages: An Empirical Investigation. *Business Process*. Retrieved from <http://www.springerlink.com/index/XN46272735MX025N.pdf>
- Rivera, J., Durán, F., and Vallecillo, A. (2009). Formal specification and analysis of domain specific models using Maude. *Simulation*. Retrieved from <http://sim.sagepub.com/content/85/11-12/778.short>
- Rodríguez, A., Fernández-Medina, E., Trujillo, J., and Piattini, M. (2011). Secure business process model specification through a UML 2.0 activity diagram profile. *Decision Support Systems*, 51(3), 446–465. doi:10.1016/j.dss.2011.01.018
- Rumpe, B., and France, R. (2011). On the relationship between modeling and programming languages. *Software & Systems Modeling*, 11(1), 1–2. doi:10.1007/s10270-011-0224-x
- Rusu, V. (2011). Embedding domain-specific modelling languages in maude specifications. *ACM SIGSOFT Software Engineering Notes*. Retrieved from <http://dl.acm.org/citation.cfm?id=1921557>
- Schmidt, D. (2012). Programming language semantics. *Information Sciences*, 1–20. Retrieved from <http://dl.acm.org/citation.cfm?id=1074733>
- Schmidt, D. A. (1997). *A METHODOLOGY FOR LANGUAGE DEVELOPMENT*. Nichols Hall, Kansas State University, Manhattan.
- Smith, GP, and Smith, G. (1992). An Object-Oriented Approach. *ukpmc.ac.uk*, (October). Retrieved from <http://ukpmc.ac.uk/abstract/CIT/692593>
- Smith, Graeme. (a) (1995.). Extending W for Object-Z. Proceeding the 9th annual Z-User Meeting. Springer. Verlag.
- Smith, Graeme. (b) (1995). of Computing A Fully Abstract Semantics of Classes for Object-Z, Formal Aspects of Computing, Springer. Verlag. 7, (3) 289-313

- Smith, Graeme. (1999). *The Object-Z Specification Language*, Springer. Verlag. I. doi:10.1007/978-1-4615-5265-9
- Smith, Graeme, and Winter, K. (2012). Incremental Development of Multi-Agent Systems in Object-Z. *lbox.itee.uq.edu.au*. Retrieved from <http://lbox.itee.uq.edu.au/~smith/pubs/sew2012.pdf>
- Stuurman, G. (2010). Action semantics applied to model driven engineering. Computer Science MSc Thesis. University of Twente.
- Taguchi, K., and Ciobanu, G. (2004). Relating Pi-calculus to Object-Z. Proceedings of Ninth IEEE International Conference on Engineering Complex Computer System. 14-16 97-106.
- Wang, H. H., Gibbins, N., Payne, T. R., and Redavid, D. (2012). A formal model of the Semantic Web Service Ontology (WSMO). *Information Systems*, 37(1), 33–60. doi:10.1016/j.is.2011.07.003
- Wang, H., Saleh, A., Payne, T., and Gibbins, N. (2007). Formal specification of owl-s with object-z. *OWL-S: Experiences an* Retrieved from <http://www.ai.sri.com/OWL-S-2007/final-versions/OWL-S-2007-Wang-Final.pdf>
- Wolterink, T. (2009). Operational Semantics Applied to Model Driven Engineering. *University of Twente MSc. Thesis*. Retrieved from <http://essay.utwente.nl/59094/>
- Woodcock, J, and Brien. S, W: a logic for Z (1991). Proceedings of Sixth Annual Z-User Meeting, University of York, 1991.

Appendix 1: Formal Semantics Description of Cloud#

o Cloud# Types

$$\text{Type} \triangleq \text{PreDefType} \cup \text{UserDefType}$$

$$\text{PreDefType} \triangleq \text{IntType} \cup \text{StringType} \cup \text{BoolType} \cup \text{VoidType}$$

$$\begin{array}{l} \boxed{\text{IntType}^{\textcircled{i}}} \quad - \quad \boxed{\text{StringType}^{\textcircled{i}}} \quad - \quad \boxed{\text{BoolType}^{\textcircled{i}}} \quad - \\ \boxed{\text{VoidType}^{\textcircled{i}}} \quad - \end{array}$$

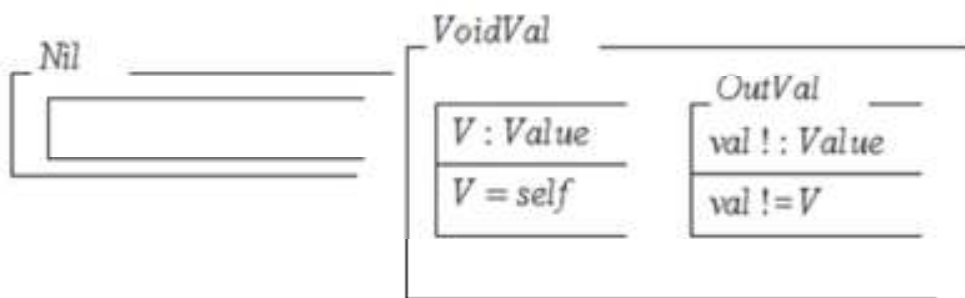
The subscript ‘ \textcircled{i} ’ on IntType specifies that there is only one instance of Integer type. It is a syntactic sugar for a system constraint: #IntType=1. Similarly, there is only one instance of Boolean type, String type, and Void type.

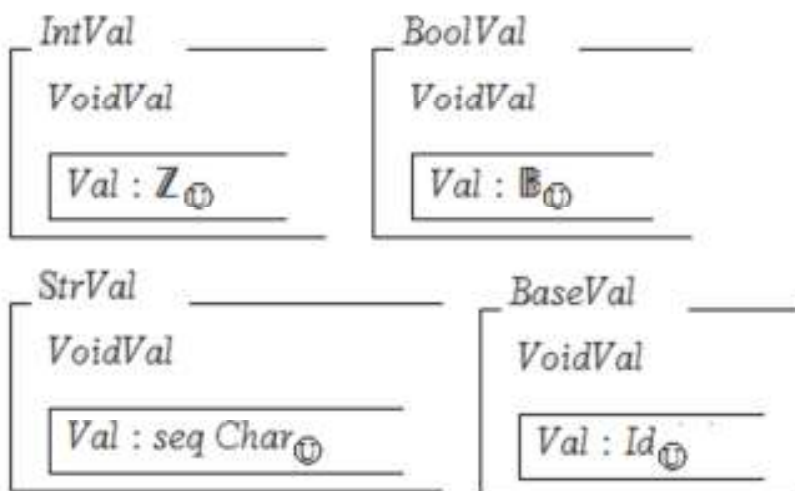
$$\text{UserDefType} \triangleq \text{BaseType} \cup \text{RecordType} \cup \text{ListType} \cup \text{TupleType}$$

$$\begin{array}{l} \boxed{\text{BaseType}} \quad - \quad \boxed{\text{ListType}} \quad - \\ \boxed{\text{id : Id}} \end{array}$$

$$\begin{array}{l} \boxed{\text{RecordType}} \quad - \quad \boxed{\text{TupleType}} \quad - \\ \boxed{\text{field : Id} \leftrightarrow \text{Type}} \quad \boxed{\text{field : seq Type}} \end{array}$$

○ **Cloud# Values**

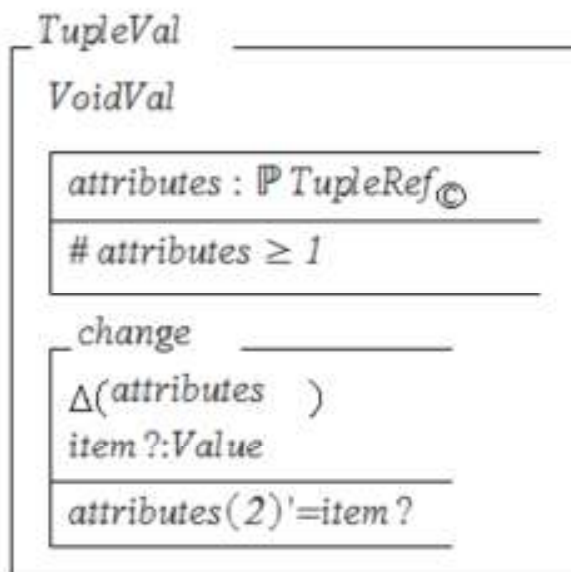
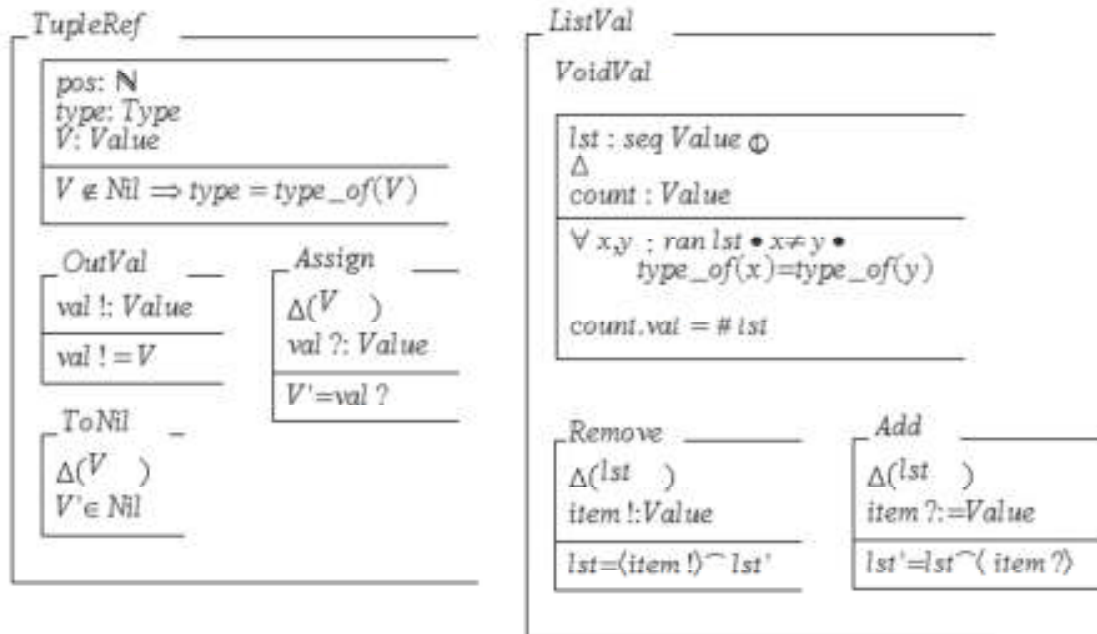
$$\text{Value} \cong \text{IntVal} \cup \text{StrVal} \cup \text{BoolVal} \cup \text{BaseVal} \cup \\ \text{RecordVal} \cup \text{TupleType} \cup \text{ListVal} \cup \text{VoidVal} \cup \text{Nil}$$


$$\text{Char} ::= \text{'a'} | \text{'b'} | \text{'c'} | \dots | \text{'z'} | \text{'1'} | \dots | \text{'9'}$$


Each instance of the class `IntVal`, `BoolVal`, `StrVal` or `BaseVal` models a particular value of its type. As represented in (Dong, 2005) the one-to-one correspondence relationship is

captured by the subscript ' $\textcircled{1}$ ' on the type of Val. Which is a syntactic sugar for a system constraint (e.g. for IntVal):

$$\forall i : \mathbb{Z} \bullet \exists_1 v : \text{IntVal} \bullet v.\text{Val} = i \wedge \forall v : \text{IntVal} \bullet \exists_1 i = \mathbb{Z} \bullet v.\text{Val} = i$$



The subscript ‘@’ on the type **Variable** or **TupleRef** refers to the non-shared object containment.

○ Mapping Function

$type_of : Value \rightarrow Type$
$\forall v : (Value \setminus Nil) \bullet$
[1] $v \in IntVal \Rightarrow type_of(v) \in IntType$
[2] $v \in BoolVal \Rightarrow type_of(v) \in boolType$
[3] $v \in StrVal \Rightarrow type_of(v) \in StringType$
[4] $v \in BaseVal \Rightarrow type_of(v) \in BaseType$
[5] $v \in ListVal \Rightarrow type_of(v) \in ListType$
[6] $v \in VoidVal \Rightarrow type_of(v) \in VoidType$
[7] $v \in RecordVal \Rightarrow$ $type_of(v) \in RecordType$ $\{a : v.attributes \bullet (a.id, a.type)\} = type_of(v).field$
[8] $v \in TupleVal \Rightarrow$ $type_of(v) \in TupleType$ $\{a : v.attributes \bullet (a.pos, a.type)\} = type_of(v).field$

○ The Variable Reference

<i>Variable</i>	
$id : Id$ $type : Type$ $V : Value$	
$V \notin Nil \Rightarrow type = type_of(V)$	
<i>OutVal</i>	<i>Assign</i>
$val ! : Value$	$\Delta(V \)$
$val ! = V$	$val ? : Value$
	$V' = val ?$
<i>ToNil</i>	
$\Delta(V \)$	
$V' \in Nil$	

○ The Arithmetic Expressions

Plus	Minus
$ \begin{array}{l} e1, e2 : \text{Exp}_{\odot} \\ \Delta \\ V : \text{Value} \\ \hline (e1.V \in \text{Nil} \vee e2.V \in \text{Nil}) \Rightarrow V \in \text{Nil} \\ \neg(e1.V \in \text{Nil} \vee e2.V \in \text{Nil}) \Rightarrow \\ \text{type_of}(V) \in \text{IntType} \\ \text{type_of}(e1.V) \in \text{IntType} \\ \text{type_of}(e2.V) \in \text{IntType} \\ V.\text{val} = (e1.V.\text{val} + e2.V.\text{val}) \end{array} $	$ \begin{array}{l} e1, e2 : \text{Exp}_{\odot} \\ \Delta \\ V : \text{Value} \\ \hline (e1.V \in \text{Nil} \vee e2.V \in \text{Nil}) \Rightarrow V \in \text{Nil} \\ \neg(e1.V \in \text{Nil} \vee e2.V \in \text{Nil}) \Rightarrow \\ \text{type_of}(V) \in \text{IntType} \\ \text{type_of}(e1.V) \in \text{IntType} \\ \text{type_of}(e2.V) \in \text{IntType} \\ V.\text{val} = (e1.V.\text{val} - e2.V.\text{val}) \end{array} $
Encryption	
$ \begin{array}{l} e1, e2 : \text{Exp}_{\odot} \\ \Delta \\ V : \text{Value} \\ \hline (e1.V \in \text{Nil} \vee e2.V \in \text{Nil}) \Rightarrow V \in \text{Nil} \\ \neg(e1.V \in \text{Nil} \vee e2.V \in \text{Nil}) \Rightarrow \\ \text{type_of}(V) \in \text{IntType} \\ \text{type_of}(e1.V) \in \text{IntType} \\ \text{type_of}(e2.V) \in \text{IntType} \\ V.\text{val} = (e1.V.\text{val} \oplus e2.V.\text{val}) \end{array} $	
Less	Equality
$ \begin{array}{l} e1, e2 : \text{Exp}_{\odot} \\ \Delta \\ V : \text{Value} \\ \hline (e1.V \in \text{Nil} \vee e2.V \in \text{Nil}) \Rightarrow V \in \text{Nil} \\ \neg(e1.V \in \text{Nil} \vee e2.V \in \text{Nil}) \Rightarrow \\ \text{type_of}(V) \in \text{BoolType} \\ \text{type_of}(e1.V) = \text{type_of}(e2.V) \\ V.\text{val} = (e1.V.\text{val} < e2.V.\text{val}) \end{array} $	$ \begin{array}{l} e1, e2 : \text{Exp}_{\odot} \\ \Delta \\ V : \text{Value} \\ \hline (e1.V \in \text{Nil} \vee e2.V \in \text{Nil}) \Rightarrow V \in \text{Nil} \\ \neg(e1.V \in \text{Nil} \vee e2.V \in \text{Nil}) \Rightarrow \\ \text{type_of}(V) \in \text{BoolType} \\ \text{type_of}(e1.V) = \text{type_of}(e2.V) \\ V.\text{val} = (e1.V.\text{val} = e2.V.\text{val}) \end{array} $
More	
$ \begin{array}{l} e1, e2 : \text{Exp}_{\odot} \\ \Delta \\ V : \text{Value} \\ \hline (e1.V \in \text{Nil} \vee e2.V \in \text{Nil}) \Rightarrow V \in \text{Nil} \\ \neg(e1.V \in \text{Nil} \vee e2.V \in \text{Nil}) \Rightarrow \\ \text{type_of}(V) \in \text{BoolType} \\ \text{type_of}(e1.V) = \text{type_of}(e2.V) \\ V.\text{val} = (e1.V.\text{val} > e2.V.\text{val}) \end{array} $	

- The Dot Expression

DotEXP
$s : \text{seq Variable}$ Δ $V : \text{Value}$
$\#s > 1 \wedge \forall i: 1..(\#s-1) \bullet$ $(s(i).V \in \text{TupleVal} \vee s(i).V \in \text{RecordVal}) \wedge$ $s(i+1) \in s(i).V.\text{attributes} \wedge V = \text{last}(s).V$

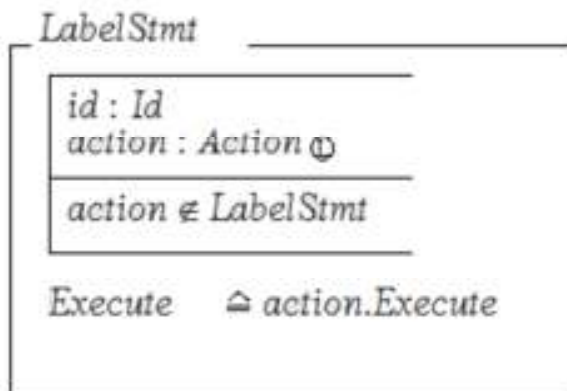
- The Assignment Statement

$\text{RetHCallStmt} ::= \{s: \text{HCallStmt} \mid s.\text{hcall}.\text{rType} \notin \text{VoidType}\}$

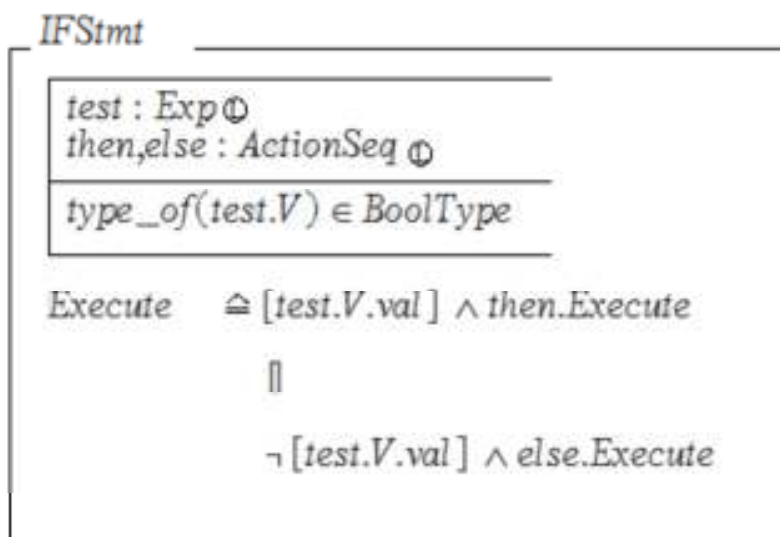
$\text{SymbolicVal} \hat{=} \text{Exp} \cup \text{RetHCallStmt}$

Assignment
$aVar : \text{Variable} \oplus$ $aSymbol : \text{SymbolicVal} \oplus$
$aVar.V \notin \text{Nil} \Rightarrow \text{type_of}(aVar.V) = \text{type_of}(aSymbol.V)$
$\text{Execute} \hat{=} [aSymbol \in \text{Exp}] \wedge aSymbol.\text{OutVal} \parallel aVar.\text{Assign}$ \parallel $[\neg (aSymbol \in \text{Exp})] \wedge aSymbol.\text{Execute} \parallel aVar.\text{Assign}$

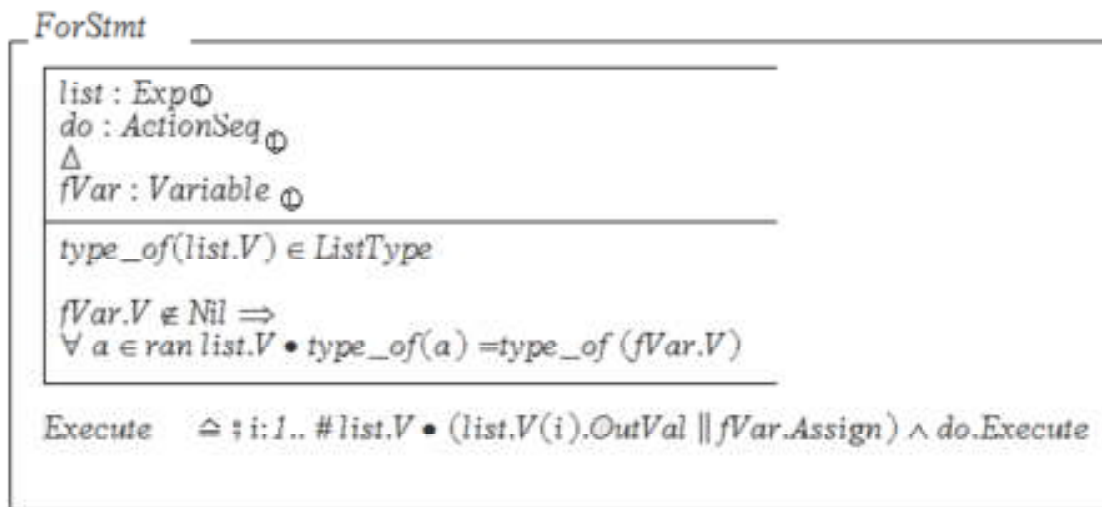
- Label Statement



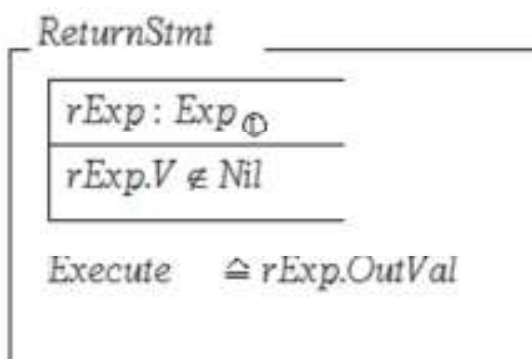
- IF Statement



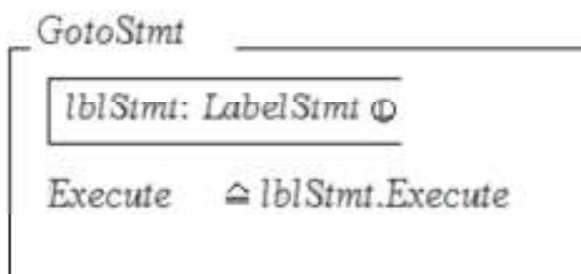
- **FOR Statement**



- **Return Statement**



- **Goto Statement**



- **Cloud# Identities**

$$\text{Id} \triangleq \text{CUnitId} \cup \text{CUnitDefId} \cup \text{TypeId} \cup \text{HCallId} \cup \text{FieldId} \cup \\ \text{ProcessId} \cup \text{VariableId} \cup \text{LabelId} \cup \text{HdefsId}$$

الملخص

يعتبر (CloudMDE) واحد من أهم المجالات البحثية في مجال تطوير البرمجيات في الوقت الحاضر. فقد استقطب اهتماما متزايدا من المجتمع البحثي. (CloudMDE) يهدف إلى ايجاد فرص تمكن الانظمة المبنية على الحوسبة السحابية من الاستفادة من التقنيات الهندسية في النمذجة و العكس بالعكس. وقد تم اقتراح لغة (Cloud#) كتطبيق على استخدام تقنيات الهندسة في النمذجة لدعم البنية التحتية لأنظمة الحوسبة السحابية. (Cloud#) هي لغة نمذجة مختصة في وصف البنية التحتية للأنظمة السحابية. (Cloud#) هي لغة من نوع (Imperative) وذات شكل نصي. تمكن هذه اللغة القدرة على التعامل مع المكونات الاساسية في البنية التحتية للأنظمة السحابية بشكل مباشر على اعتبارهم اجزاء اساسية من اللغة. علاوة على ذلك، هذه اللغة تدعم التزامن والاستجابة للمؤثرات الخارجية. حتى الآن يتوفر لهذه اللغة الوحدات البنائية لبناء جملة مجردة (المفردات) و الضوابط التي يجب اتباعها لكتابة جمل صحيحة في اللغة بالاضافة الى وصف للمعنى الوظيفي للوحدات البنائية في هذه اللغة. ومع ذلك، هذه اللغة تفتقر إلى تعريف المعنى الوظيفي بشكل رياضي. في هذه الأطروحة، تم وصف المعنى الوظيفي للوحدات البنائية في اللغة بشكل رياضي. وقد تم استخدام اللغة الرياضية (Object-Z) لوصف لغة (Cloud#). لقد تم وصف مفردات اللغة في اطار واحد وموحد بحيث ان وصف المفردات والضوابط لكتابة جمل صحيحة بالاضافة الى المعنى الوظيفي للمفردات تم وضعة في وحدة واحدة وهذا يساعد على سهولة قراءة الوصف، بل و يساعد ايضا على اجراء التعديلات على الوصف الرياضي بطريقة سهلة في حالة تطوير او تحديث اللغة. ايضا مثل هذا الوصف يمكن من اعادة استخدام بعض اجزائه لتعريف لغات اخرى. من ناحية أخرى، تم التحقق من عدم وجود اي تناقضات في لغة (Cloud#) و ذلك باستخدام احد ادوات التحقق في لغة (Object-Z). حيث تم اخذ أحد نماذج لغة (Cloud#) وتم تحميله مع التعريف الرياضي الذي تم انجازه في هذه الاطروحة الى أحد أدوات لغة (Object-Z) حيث لم تظهر النتائج اي تناقضات في هذه اللغة مما يدل على ان هذه اللغة سليمة وخالية من الاخطاء التصميمية.



السيمانتك الشكلي للغة الكلود شارب

إعداد

يحيى مصطفى عبد الرحمن

المشرف

د. مراد معوش

قدمت هذه الرسالة استكمالاً لمتطلبات الحصول على درجة الماجستير في تخصص علم الحاسوب

عمادة البحث العلمي والدراسات العليا

جامعة فيلادلفيا

كانون الثاني، ٢٠١٣