

**SPECIFICATION LANGUAGE DESIGN CONCEPTS:
AGGREGATION AND EXTENSIBILITY IN CoreASM**

by

Mashaal Anwar Memon
B.C.B., Brock University, Ontario, Canada, 2001

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

© Mashaal Anwar Memon 2006
SIMON FRASER UNIVERSITY
Summer 2006

All rights reserved. This work may not be
reproduced in whole or in part, by photocopy
or other means, without the permission of the author.

APPROVAL

Name: Mashaal Anwar Memon
Degree: Master of Science
Title of thesis: Specification Language Design Concepts: Aggregation and Extensibility in CoreASM

Examining Committee: Dr. Lou Hafer
Chair

Dr. Uwe Glässer, Senior Supervisor

Dr. Thomas Shermer, Supervisor

Dr. Robert Cameron, SFU Examiner

Date Approved:

April 21st, 2006



**SIMON FRASER
UNIVERSITY** library

DECLARATION OF PARTIAL COPYRIGHT LICENCE

The author, whose copyright is declared on the title page of this work, has granted to Simon Fraser University the right to lend this thesis, project or extended essay to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users.

The author has further granted permission to Simon Fraser University to keep or make a digital copy for use in its circulating collection, and, without changing the content, to translate the thesis/project or extended essays, if technically possible, to any medium or format for the purpose of preservation of the digital work.

The author has further agreed that permission for multiple copying of this work for scholarly purposes may be granted by either the author or the Dean of Graduate Studies.

It is understood that copying or publication of this work for financial gain shall not be allowed without the author's written permission.

Permission for public performance, or limited permission for private scholarly use, of any multimedia materials forming part of this work, may have been granted by the author. This information may be found on the separately catalogued multimedia material and in the signed Partial Copyright Licence.

The original Partial Copyright Licence attesting to these terms, and signed by this author, may be found in the original bound copy of this work, retained in the Simon Fraser University Archive.

Simon Fraser University Library
Burnaby, BC, Canada

Abstract

Abstract State Machines (ASMs) are a proven methodology for the precise high-level specification of formal requirements in early phases of software design. Many extensions to ASMs have been proposed and used widely, including Distributed ASMs, Turbo ASMs, Gurevich's partial updates, and syntactically convenient rule forms. This, coupled with the fact that ASMs do not bind the user to any predetermined data types or operators, allows for extreme flexibility in exploration of the problem space. Striving to provide this same level of freedom with executable ASMs, the CoreASM engine and language have been designed with syntactic and semantic extensibility in mind. We formally specify extensibility mechanisms that allow for language augmentation with arbitrary data structures supporting simultaneous incremental modification, new operators, and additional language syntax. Our work is a major step toward providing an environment suitable for both further experimentation with ASMs and for the machine-aided creation of robust software specifications.

Keywords: abstract state machine (ASM), executable ASMs, specification language extensibility, aggregation, partial updates

*To my parents Anwar and Najma.
Your love and support have made me what I am.*

“If you always put limit on everything you do, physical or anything else. It will spread into your work and into your life. There are no limits. There are only plateaus, and you must not stay there, you must go beyond them.”

— Bruce Lee

Acknowledgements

There are many people who have touched my life during my Masters studies, all playing a part in pushing me forward towards my final objective. These people deserve much more than mere acknowledgement. My mentioning them here serves only as a minute gesture relative to the impact they have all had on me.

I thank my supervisor Uwe Glässer for his understanding and guidance. The CoreASM project is a result of his vision and I am grateful that he was able to impart this vision in me.

I am indebted to Michael Letourneau for the help he gave me from beginning to end, which came in many forms. He was an advisor, a software tester, a proof-reader, a great roommate, but most importantly a friend. I am glad he was around.

Many days (and some nights) in the Software Technology lab involved intense discussions with Roozbeh Farahbod on CoreASM, ASMs, and other random (often not so scholarly) topics. For his insightful arguments, his good advice, and the well deserved distraction he provided me, I give him my heartfelt thanks.

I could not have made it to the end without the constant nurturance and support given to me by Melissa Gonsalves. She was always sure of my potential even at times when I was not.

Without my parents' dreams for my future and efforts to shape me, I could not have reached this point in my life. I would not have been able to decipher, nor stay focused on, my goals were it not for my brother Da'anish. The appreciation I have, for the unlimited love and encouragement given to me by my family, simply cannot be expressed in words.

Finally, I give thanks to true friends made on this journey: Eric Evangelista, Mayu Ishida and Vipul Mehra. Their presence and company alone made the years fly by.

Contents

Approval	ii
Abstract	iii
Dedication	iv
Quotation	v
Acknowledgements	vi
Contents	vii
List of Tables	xi
List of Figures	xii
List of Specifications	xiv
List of Abbreviations and Acronyms	xv
1 Introduction	1
1.1 Motivation and Related Work	3
1.2 Objective and Significance	5
1.3 Organization of Thesis	6
2 Abstract State Machines	7
2.1 Basic Abstract State Machine	7
2.1.1 Non-determinism	10

2.1.2	Parallelism	11
2.1.3	Function/Relation Classification and Environment	12
2.2	Turbo ASMs	14
2.2.1	Iterative Composition	15
2.3	Distributed Abstract State Machines	15
2.4	Additional Rule Forms	16
2.4.1	Temporary Variables	17
2.4.2	Return Values	17
2.4.3	Substitution	18
2.5	Control State ASMs	19
2.6	Background Classes	21
2.7	Partial Updates	22
2.7.1	General Mathematical Framework	23
2.7.2	Turbo ASMs and Sequential Composition	24
2.7.3	Rule Forms	24
2.8	Modelling Computer Languages	25
2.9	Notational Conventions	26
3	Compiler Theory Preliminaries	28
3.1	Interpreter vs. Compiler	28
3.2	Language Definition	29
3.3	Input Representation and Evaluation	30
3.4	Operator Classification and Precedence	32
3.5	Types and Semantic Analysis	34
3.5.1	Dynamic vs. Static Typing	35
3.5.2	Strong vs. Weak Typing	35
4	CoreASM Overview	37
4.1	Engine Architecture	38
4.1.1	CoreASM Components	40
4.1.2	Engine Life-cycle	42
4.1.3	The Kernel and Plug-ins	50
4.2	The CoreASM Language	51
4.2.1	Notation	51

4.2.2	Kernel Interpreter	56
4.2.3	Extension Interpretation	57
5	Distributed Incremental Change with Aggregation	62
5.1	Aggregation and Incremental Updates	63
5.1.1	Rules and Their Side Effects	63
5.1.2	Update Instruction Notation	64
5.1.3	A CoreASM Step	65
5.1.4	Responsibility for Aggregation	68
5.1.5	Basic Update Aggregator	69
5.1.6	Plug-in Aggregation Consistency	70
5.1.7	Aggregation Algorithms Provided	71
5.1.8	Aggregation Phase Consistency	71
5.2	Turbo ASMs and Sequential Composition	73
5.2.1	Update Multiset Composition	73
5.2.2	Basic Update Composer	75
5.2.3	Interpretation of the Seq-rule	76
5.3	Set Plug-in	77
5.3.1	Background Extension	77
5.3.2	Operators Extension	83
5.3.3	Rule Extension	85
5.3.4	Aggregation Algorithm	86
5.3.5	Composition Algorithm	90
5.4	Summary	93
6	Operator Evaluation and Language Additions	94
6.1	Operator Extension and Evaluation	95
6.1.1	ASM Type Conventions	95
6.1.2	Operators in ASMs	96
6.1.3	CoreASM Type Conventions and Operator Extension	96
6.1.4	Significance of Operator Evaluation in CoreASM	98
6.1.5	CoreASM Operator Evaluation	98
6.2	Parser Extensibility	103
6.2.1	The CoreASM Language Dependence on Specifications	103

6.2.2	Dynamic Grammar	105
6.3	Summary	106
7	Conclusion and Accomplishments	107
7.1	Implementation and Project Involvement	108
7.2	Future Work	110
A	CoreASM Component and Plug-In Interfaces	111
A.1	Abstract Storage	111
A.1.1	Elements of the Superuniverse	113
A.1.2	Element Enumerability	113
A.2	Set Plug-In	114
B	How to Shoot Yourself in the Foot with CoreASM	118
	Bibliography	119
	Index	123

List of Tables

4.1	Abbreviations and their meanings in syntactic PA rules.	54
4.2	Examples of how PA rule notation is translated into ASM rules.	55

List of Figures

2.1	Classification of ASM functions, relations, and locations.	13
2.2	Control State ASM : Conditional execution of rule with conditional control state change.	19
2.3	Control State ASM : Execution of single rule with conditional control state change.	20
2.4	Control State ASM : Execution rule and control state change based on truth of single condition.	20
2.5	Control State ASM : Sequential execution of rules.	20
3.1	A simple expression grammar.	30
3.2	Example parse tree.	31
3.3	Example abstract syntax tree.	31
3.4	Evaluated abstract syntax tree.	32
3.5	AST for left and right associative operators.	33
3.6	AST resulting from OPG.	33
3.7	Expression grammar, now an OPG, taking into account operator precedence.	34
4.1	Architectural Overview of CoreASM Engine	38
4.2	Layers and Modules of the CoreASM Engine	40
4.3	Sample annotated AST	41
4.4	Control State ASM of Engine Initialization	43
4.5	Control State ASM of Loading of CoreASM Specification	44
4.6	Control State ASM of a STEP command: Control API Module	44
4.7	Control State ASM of a STEP command : Scheduler	45
4.8	Control State ASM of a STEP command : Abstract Storage	47

4.9	Control State ASM of a STEP command : Interpreter	48
5.1	Revised control state ASM of a Step command with Aggregation : Scheduler	67
5.2	Revised control state ASM of a Step command with Aggregation : Abstract Storage	67
6.1	An example use of use directives in a CoreASM specification.	104
6.2	The kernel grammar structure showing GEPs.	105

List of Specifications

4.1	The ExecuteTree rule in the Interpreter module.	49
4.2	The KernellInterpreter rule in the Interpreter module.	56
4.3	The PA rule for interpretation of the seq-rule.	59
4.4	PA rule depicting semantics of the equivalence operator.	60

List of Abbreviations and Acronyms

ASM	Abstract State Machine
AST	Abstract Syntax Tree
DASM	Distributed Abstract State Machine
FIFO	First In First Out
GEP	Grammar Extension Point
LA	Left Associative
LHS	Left-hand Side
OB	Operator Behaviour
OD	Operator Definition
OPG	Operator Precedence Grammar
OP	Operator Precedence
PA rule	Pattern-action Rule
RA	Right Associative
RHS	Right-hand Side

“Begin at the beginning, . . .”

— King of Hearts, Alice in Wonderland

Chapter 1

Introduction

The CoreASM project is a research effort aiming to specify and implement an extensible execution engine for a language that is as close to the mathematical definition of pure Abstract State Machines (ASM) [35] as possible. This thesis presents work done to ensure that the CoreASM engine and language remain as flexible as the ASM formalism on which they are based. We will use ASMs to formally specify how extensibility mechanisms for distributed incremental modification of data structures, operators, and language syntax have been incorporated into the engine. This specification has been used to establish the basis for implementing these features in CoreASM, further bolstering the already strong reputation of ASMs as a powerful formal specification method.

In early phases of system design, the transformation of informal requirements into precise specifications is invaluable, as deficiencies hidden in informal requirements can be found and dealt with. Through analysis of key design choices and their implications, software designs can be more thoroughly validated prior to implementation. ASMs provide a framework for precise semantic modelling of functional requirements, a framework that is built on a rigorous mathematical foundation [5, 36]. Because the ASM formalism allows for a high level of abstraction, it is an effective tool for gaining a clear understanding of design problems and their solutions. It is also an attractive choice for system specification, because the ASM language uses a syntax akin to that of imperative programming languages and so it will be familiar to those involved in implementation.

The product of formalizing a system using the ASM paradigm is a representative ASM *ground model* [9], a semantic ‘blueprint’ which documents the key system requirements,

that is visible and inspectable by analytical means and empirical techniques. The versatility of ASMs has been proven through their practical application, which has resulted in ground models for many programming and system specification languages (see Section 4), communication architectures [33, 34], and embedded control systems [14, 3, 13].

Many semantic extensions, that make modelling with ASMs more convenient, have been defined and adopted over the years of ASM study and use in academia and industry: Distributed Abstract State Machines (DASM) [15, Chapters 5-6] facilitate the modelling of multi-agent distributed systems; Turbo ASMs [15, Chapter 4] facilitate highly structured modelling; the notion of partial updates [38, 39] allow for a data structure instance to be simultaneously modified by different parts of a system. These extensions, coupled with the freedom to define and then use data types and operators specific to the system being modelled, give a systems architect great flexibility.

ASMs provide a simplified view of system behaviour as the evolution of abstract states over discrete time steps, where abstract states are represented as variants of first-order (Tarski) structures. These simple state transition systems can in principle be simulated using a computer, resulting in executable ASM specifications. Making specifications machine-executable has many advantages:

- The behaviour of a model can be easily observed and explored under different conditions, allowing unwanted or unexpected behaviours to be found and eliminated.
- The specifications executed have a single well-defined semantics which is enforced by the execution environment, resulting in unambiguous interpretation by all parties.
- Further guarantees on the correctness of a specification can be given, since real-time interaction and inspection with analytical and empirical techniques becomes feasible (e.g. user interaction and visualization with graphical mockups and GUIs [52], automated regression testing and test case generation [31], and symbolic model checking [19]).

In general, machine assistance makes the design, validation, and implementation of practical systems more feasible.

1.1 Motivation and Related Work

Since the inception of ASMs, many executable variants have been developed; the most popular and advanced are: AsmL (ASM Language) [44], the ASM Workbench [18], XASM (eXtensible Abstract State Machines) [2], and AsmGofer [48].

While the CoreASM project is not the first attempt to make ASMs executable, both the engine and language have many novel features which differentiate them from their predecessors [23]. Underlying the CoreASM project has been a *core ideology* we define here, that provides motivation for the endeavour and guides CoreASM development:

1. *The preservation of pure ASM semantics:* Other variants have had concessions made with respect to their semantics in the form of strict typing conventions and object oriented characteristics. CoreASM semantics should not deviate from the original pure mathematical definition of ASMs. While new semantic and syntactic additions are welcome, these cannot not be included at the cost of modifying existing semantics.
2. *Ensuring freedom through extensibility:* While other variants do allow for a certain level of customizability, all have many restrictions including the data types available, operators available, and syntax which can be used. ASMs do not have such restrictions, and the CoreASM language should similarly allow for the same amount of freedom; it accomplishes this via a robust plug-in architecture.

Motivated by the second tenet of the CoreASM core ideology, this thesis focuses primarily on three different extensibility mechanisms for the CoreASM engine and language.

In this work we present a comprehensive formal specification of a framework which allows for a data structure in CoreASM to be incrementally modified by multiple-agents simultaneously. Consider two clients adding and removing portions of the same file using a central source configuration management system (SCM)¹ such as CVS or Perforce²; when modifications are made concurrently by the clients, the SCM must ensure that all changes made are aggregated together resulting in a single consistent global change to the file. To incorporate similar functionality in ASMs, all incremental modifications must be collected and

¹SCMs are client/server systems used to facilitate concurrent team-based development and to provide file revision history in software projects. CVS is a well known and free SCM and Perforce is a commercial variant.

²Perforce Software, <http://www.perforce.com>

aggregated together to determine the resultant change to be applied to the data structure and to ensure that incremental modifications do not conflict semantically with each other. Before Gurevich's work on partial updates [38, 39], ASMs could not handle simultaneous incremental modification (vs. overwriting) of individual memory locations. His work has resulted in a general mathematical framework providing the theoretical support for the distributed simultaneous modification of data structures within ASMs. The incorporation of this framework into ASMs involves amendments to the classic process of ASM state transition which do not change or violate, but rather build upon, ASM semantics. This framework has been incorporated into *AsmL*³, resulting in the first executable ASM specification language allowing for incremental modification of data structures. *CoreASM* will be the second such language. However, unlike *AsmL*, this functionality will not be constrained only to data types which come with the engine. The *CoreASM* language can be easily extended with third-party plug-ins introducing new data types along with aggregation methods relevant to their data structures. We showcase this functionality by defining a data type for sets which supports incremental change.

We also formalize the process used in *CoreASM* to execute the correct variant of an overloaded operator. In most programming languages, data type requirements of operators are known at compile-time or run-time and are used to choose the appropriate operator variant [1, Section 6.5]. All other executable ASM languages are strictly typed and as a result, choose operator variants in much the same way. Adherence to the first tenet of the project ideology has steered *CoreASM* away from the adoption of a strictly-typed language model, and adherence to the second tenet has led to the support of extensions allowing new operators to be defined for use. The novelty of our operator evaluation approach is that selection of operator variant need not depend on data type information.

Lastly, we discuss the extensibility mechanism in the engine which allows for the extension of the language syntax, and facilitates the addition of new operators, literals for new data types, etc. We show that the language at any time is based on the *CoreASM* specification being executed, and how the grammar of the language is structured to allow for syntactic extension. To the best of our knowledge, no other computer language includes a plug-in based system that allows such extension. In programming languages, the common approach for dealing with syntax extensions is to use macro languages to express them; this

³Gurevich spearheaded the effort to create *AsmL* during his tenure with the Foundations of Software Engineering group at Microsoft Research, Redmond.

approach involves a preprocessing step before compilation (or interpretation) where the syntax is extended according to macros defined in the source file. Run-time grammar extension using such macro languages has been explored [16], however we have found no discussion of grammar structure and how it facilitates syntax extensibility. In CoreASM the ability to extend the language syntax is essential for realizing the flexibility we aim to provide.

1.2 Objective and Significance

The freedom and flexibility that ASMs provide in modelling are key to their usefulness in system specification. The main objective of our work is to infuse into CoreASM the same level of flexibility by:

- Integrating Gurevich's notion of partial updates into a CoreASM step. We call our method of accommodating simultaneous incremental change into CoreASM, *aggregation*.
- Allowing for future extension of the engine with arbitrary data types that permit aggregation.
- Formalizing a method of operator evaluation that allows for the definition of arbitrary overloaded operator behaviours, but that is not bound to selection of overloaded behaviour based on operand data type.
- Formally describing our pragmatic approach to extending the CoreASM language with additional syntax.

While we have used Gurevich's mathematical framework as an inspiration for aggregation in CoreASM, the move from theory to practice is not simple. Although the partial update framework has been incorporated into the AsmL compiler, no specification of or documentation on the approach taken to its implementation could be found at the time of writing. In our attempt to incorporate a similar yet extensible framework in CoreASM, we have not only realized our goal of forward compatibility of aggregation with arbitrary data structures, we have also produced a solid ASM ground model for executable ASM semantics with support for simultaneous incremental change.

Our efforts towards CoreASM extensibility ensure that it imposes few restrictions which stifle freedom of experimentation and exploration of the problem space. Through our work,

data types, operations and language additions specific to each problem's domain can be added as needed. The development of our extensible specification execution environment will be beneficial both to researchers, who will be able to test future extensions to ASMs, and to industry, who will use the engine and its extensions to make more precise software specifications resulting in more reliable software.

The result of the formalization of our extensibility mechanisms is a specification that can be validated through machine-aided simulation and testing using the CoreASM engine once it is complete.

1.3 Organization of Thesis

We begin by providing an overview of ASMs and useful extensions including DASMs, Turbo ASMs and Gurevich's partial updates in Chapter 2. Chapter 3 is a brief primer on compiler theory concepts required to fully understand the CoreASM engine and language specification. Previous work done on formalizing functional requirements of the CoreASM engine architecture and language is presented using a high-level ASM specification in Chapter 4; in subsequent chapters we shall build on this work. In Chapter 5 we describe how the concept of Gurevich's partial updates is incorporated into the CoreASM engine through aggregation while facilitating future extensibility with arbitrary data types. We then provide a full specification of the Set Plug-in which uses the newly introduced extensibility mechanism for distributed incremental updates on set data structures, as well as other extensibility mechanisms for defining the Set data type and set related literals, operators and rule forms. Chapter 6 describes the difficulties associated with overloaded operators and their evaluation in CoreASM, and ends with a description of how the CoreASM language syntax is made extensible. Chapter 7 concludes the thesis with a summary of our work, a description of efforts made towards the implementation of CoreASM, and directions for future work.

*“I’ve been doing a lot of abstract painting lately,
extremely abstract. No brush, no paint, no
canvas, I just think about it.”*

— Steven Wright

Chapter 2

Abstract State Machines

Both the executable specifications interpreted by CoreASM, and the CoreASM model presented are based on the *Abstract State Machine* (ASM) paradigm. We will, in this chapter, briefly describe ASM concepts from their most basic foundations, to many novel and necessary additions. In the first section we will introduce Basic ASM concepts, which are central to the ASM paradigm, and common to all variants; this will include facilities for *parallelism* and *non-determinism*. This will be followed by a description of the Turbo ASM extensions which allow for structured composition as well as *sequential* and *iterative* execution within models. *Distributed* Abstract State Machines (DASM), which facilitate the modelling of multi-agent distributed systems, will be introduced in the third section. With a firm understanding of these ASM variants, we will further introduce additional rule forms, Control State ASMs, Background Classes, and the concept of Partial Updates. We end this chapter with a brief discussion of the applicability of ASMs to computer language development and a description of notational conventions used to present our ASM models.

While our introduction to ASMs is sufficient for a rich understanding of the work presented in this thesis, it can by no means be considered anything more than an informal yet intuitive description of key concepts. We will direct the reader to original literature with more comprehensive definitions where appropriate.

2.1 Basic Abstract State Machine

For a detailed description of Basic ASMs, the reader is directed to [35] and [15, Chapter 1-3]. A *Basic* ASM M consists of a *program* P_M , a set of *states* S_M , and a collection of

initial states I_M where $I_M \subseteq S_M$.

Each state $S \in S_M$ of an ASM M can be interpreted with the same *vocabulary* (or *signature*) Υ , a finite collection of function and predicate names each with a fixed arity. Υ_M will always contain these static function names: the equality sign ($=$), nullary function names *true*, *false*, *undef* and the names of the usual Boolean operations (\wedge, \vee, \neg). Notice that states of ASMs are not unlike first-order structures in mathematical logic.

Any state S is a nonempty set U , known as the *superuniverse* or *base set*, along with the interpretations of Υ on U . An n -ary function name interpreted as a function from U^n to U , is called a *basic function* of S . An n -ary relation name interpreted as a relation from U^n to $\{\text{true}, \text{false}\}$, is referred to as a *basic relation* or *predicate* of S . A *location* l consists of an n -ary function or predicate f and any n -ary tuple \bar{x} consisting of elements from U ; so l is equivalent to $f(\bar{x})$. The concept of locations provides the ASMs with memory, while allowing for abstraction from the methods of memory addressing and object referencing. In any S a given location holds a value, that being a single element of U . All functions and relations are total; the default value of all basic functions is *undef* and of all basic relations is *false*. The logic names *true* and *false*, along with *undef* are interpreted to be distinct elements of U , and the Boolean operations behave as they normally would on the values *true* and *false*.

Special unary predicates defined over U allow us to view any S as a many-sorted structure; such predicates can be viewed as special *universes* or *domains*. Note that universe membership may overlap, as is the case with all universes and the superuniverse.

ASM *terms* represent a location in a state. Given that f is an n -ary function name and t_1, t_2, \dots, t_n are terms, then $f(t_1, t_2, \dots, t_n)$ is a term as well. Nullary functions or predicates are also called *variables*.

Every ASM *rule* produces a (potentially empty) set of *updates*, each update consisting of a location and a new value: $u = \langle l, v \rangle$. An ASM program consists of a single rule which is interpreted in the current state of the machine S_1 ; as will be seen, rules may be composed of other rules.

The most basic rule of ASMs, the **update**-rule, assigns a value to a given location and is of the form:

$$f(t_1, t_2, \dots, t_n) := t_0$$

Here t_0, \dots, t_n are evaluated in the current state S_1 resulting in values $v_0, \dots, v_n \in U$; this

rule produces an update of the form $\langle f(v_1, v_2, \dots, v_n), v_0 \rangle$. The update represents an action to be taken, namely that in the next state S_2 of the ASM machine, the current value of the location $f(v_1, v_2, \dots, v_n)$ should be replaced with a new value v_0 .

An **ifelse**-rule, being a *conditional rule*, has the form

```

if  $t$  then
   $R_1$ 
else
   $R_2$ 

```

which if the term t evaluates to *true* executes the rule R_1 , and otherwise executes R_2 .

When describing an algorithm, often times it is necessary to have resources available facilitating the introduction of new elements. In ASMs, the **import**-rule dynamically allocates fresh elements from a set called the *reserve*.

```

import  $e$ 
   $R[e]$ 

```

The **import**-rule selects an element from the reserve, points the temporary function e to it, removes it from the reserve, and initiates execution of the rule R . We use the notation $R[x]$ to mean that x occurs freely in transition rule R . Only R has access to (or is in scope of) e , and may use the new element. It is important to note that neither the reserve set nor any of its elements can be directly accessed. Also note that although the reserve is not required to be present in a vocabulary Υ , when fresh elements are required by program P_M of ASM M , it is assumed that Υ_M does include the function *Reserve*; all members of the reserve are then present in U_M , and hence in every state S [35]. Similarly, a universe can be extended with a new element dynamically, using the **extend**-rule:

```

extend DOMAIN with  $e$ 
   $R[e]$ 

```

Again an element from the reserve is selected and assigned to e , followed by the execution of R . However, the rule also creates an update of the form $\langle \text{DOMAIN}(e), \text{true} \rangle$, making the new element a member of the domain DOMAIN.

When creating abstract specifications, one may wish to create placeholders for future refinement. This is accomplished with the **skip**-rule

skip

which produces an empty set of updates, effectively doing nothing.

Facilitating modularization and maintenance of organized specification, **call-rules** or **named-rules** can be defined. A *rule declaration* for a rule name **NamedRule** of arity n has the form:

$$\mathbf{NamedRule}(x_1, \dots, x_n) \equiv R[x_1, \dots, x_n]$$

When any **named-rule** is declared in a specification, it is executed with a call:

$$\mathbf{NamedRule}(t_1, \dots, t_n)$$

This results in an execution of R , with all instances of free variables x_i replaced by the corresponding values v_i obtained by evaluating all terms t_1, \dots, t_n in the current state.

2.1.1 Non-determinism

To describe processes at high levels of abstraction, non-determinism is beneficial. Details of scheduling of subprocesses in execution can be hidden from the user. Basic ASMs provide the **choose-rule** to facilitate this:

$$\mathbf{choose } e \in C \\ R[e]$$

An element which is a member of C is chosen non-deterministically, this element is assigned to e , and then R is then executed. If there is no such element, nothing is done.

The **choose-rule** can both be augmented with a guard, and/or an **ifnone** clause:

$$\mathbf{choose } e \in C \mathbf{ with } g[e] \\ R_1[e] \\ \mathbf{ifnone} \\ R_2$$

Before rule R_1 can be executed, the element e chosen from C must satisfy the guard condition g ; Essentially e is chosen from $C_g \subset C$ where $C_g = \{x \mid x \in C, g(x) = \text{true}\}$. With the

ifnone clause in place, R_2 is executed iff no e can be chosen (e.g. if C is empty, or no element of C satisfies the guard).

2.1.2 Parallelism

Every ASM M has one *main rule* or program P_M . This main rule may only result in the execution of multiple rules with the use of one of the, soon to be introduced, parallel constructs. As such, when the main rule is executed, all of its child rules are executed in parallel. Simultaneous execution has two main benefits for high-level design and specification [15]:

1. It provides a convenient way to abstract from sequentiality where it is irrelevant.
2. It allows for the local description of a global state change: a transition to the next state is the result of updates created independently of each other.

The **block-rule** or **par-rule** provides the following parallel rule notation:

$$\begin{array}{l} \mathbf{par} \\ R_1 \\ \vdots \\ R_n \end{array}$$

where all rules R_1, \dots, R_n are executed simultaneously. Often times the keyword **par** is dropped, with the implication that all rules occurring one after the other are executed simultaneously.

The **forall-rule**, allows for an enhanced notion of parallelism, having the form

$$\begin{array}{l} \mathbf{forall} \ e \in C \\ R[e] \end{array}$$

where R is executed simultaneously for all $e \in C$.

In a similar fashion to **choose**, the **forall-rule** can be extended with a guard:

$$\begin{array}{l} \mathbf{forall} \ e \in C \ \mathbf{with} \ g[e] \\ R[e] \end{array}$$

The rule R will only be executed for those elements of C which satisfy the guard g .

A Transition System

The result of the execution of the main rule of M is a transition or *move* from one state S_i to S_{i+1} . All transition rules executed by evaluation against S_i result in updates which are collected into the *update set* Δ . If the update set is considered to be *consistent*, the computation of this *step*, done by *applying* (also known as *firing*) the update set, yields the next state S_{i+1} . To be called consistent, an update set must not contain two different updates to the same location (i.e. no two updates where $\{\langle l, v \rangle, \langle l, v' \rangle\} \in \text{updateSet}$ with $v \neq v'$). An *inconsistent* update set creates ambiguity in the interpretation of what state should follow, precluding the possibility of a transition, and resulting in the machine halting.

A *run* of an ASM M is a sequence of states beginning from an initial state $S_1 \in I_M$. A transition between states $S_i \rightarrow S_{i+1}$ will be denoted as T_i , and the update set causing said transition will be denoted Δ_i .

An ASM can model reactive systems which cycle their computation step indefinitely, but for the case of systems which terminate when complete, various termination criteria can be selected [15]:

- No rule is applicable any more.
- The machine yields an empty update set: rules may be executed, but no updates are produced by their execution.
- The state does not change any more: updates are produced, but are redundant.

2.1.3 Function/Relation Classification and Environment

Every ASM may interact with its environment or other agents. By the term *environment* we mean anything not under the control of current agent being executed.

Figure 2.1¹ depicts the hierarchy of functional classification in ASMs. All functions (and relations) can be split into two types: *basic* functions and *derived* functions. Basic functions are those whose individual locations may not necessarily be defined by a mathematical function. Derived n-ary functions are functions whose value at any given location can be defined by some mathematical formula, typically with n corresponding free variables; derived functions are not updatable, but may rely on basic function locations.

¹This figure was reproduced from Figure 2.4 on page 33 of [15] with kind permission of Springer Science and Business Media.

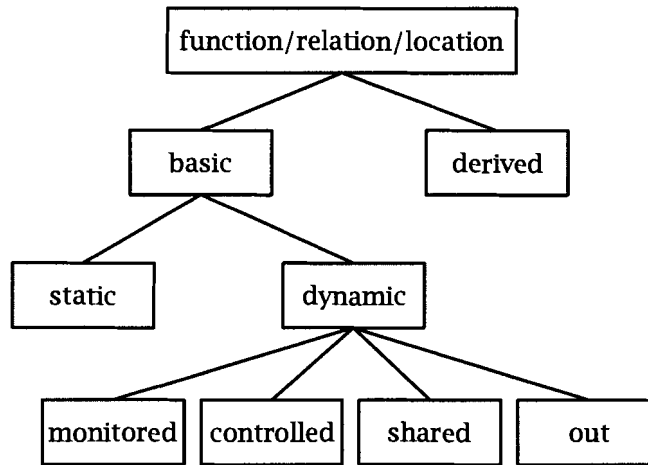


Figure 2.1: Classification of ASM functions, relations, and locations.

Basic functions can be further split into two different classes, *static* and *dynamic*. Static functions are those whose value does not change during the run of M and so whose arguments never depend on any single state of M . Clearly one would not find a static function on the left-hand side of an **update**-rule.

Dynamic functions, on the other hand, may change as a consequence of updates or by the intervention of the environment. This class can be further divided into four subclasses based on who can modify them and who can read their locations.

- **Monitored (in)** - can only be read by M , and written to only by the environment. Such a function will not be found on the left side of an **update**-rule.
- **Controlled** - can be written to only by M . This class of functions may be found on either side of an **update**-rule. If the function f is controlled, and the update $\langle f(x), v \rangle \in \Delta_i$ is applied in T_i , then one can be sure the $f(x) = v$ in S_{i+1} .
- **Shared** - can be read from and written to by M or the environment. Such a function may be found on either side of an **update**-rule. If the function f is Shared, and the update $\langle f(x), v \rangle \in \Delta_i$ is applied in T_i , then one cannot be sure the $f(x) = v$ in S_{i+1} ; the environment may have updated the location after Δ_i was applied but before T_{i+1} was initiated.

- **Out** - can written to by only M , and read only by the environment. An out function would only be found on the left side of an **update-rule**.

2.2 Turbo ASMs

While convenient for high level specification of simple models, Basic ASMs lack in syntax facilitating practical composition and support for structuring principles. *Turbo* ASMs, an extension to the basic paradigm, offer building blocks such as *sequential composition* and *iteration*. For more in depth coverage of this subject, the reader is directed to [15, Chapter 4].

With Turbo ASMs, a step is still considered to happen instantaneously, however each step may contain elementary actions or *micro steps*, which are executed in a fixed order. The internals of these subcomputations are hidden by compressing them into a single step of the machine.

Sequential Composition

The **seq**-rule allows for sequential execution of multiple rules with the the Basic ASM concept of simultaneous updates of locations in the global state of the machine:

$$R_1 \text{ seq } R_2$$

First R_1 is executed on state S_1 , resulting in updates Δ_1 . If Δ_1 is consistent then it is fired producing a temporary state S_2 on which R_2 is executed. If Δ_2 is also consistent, then the two update sets must be sequentially composed (via \oplus) into a single set of updates Δ_{seq} that being the set of updates produced by the rule in this ASM step. Thus

$$\Delta_{seq} = \Delta_1 \oplus \Delta_2 = \{\langle l, v \rangle \in \Delta_1 \mid l \notin Locs(\Delta_2)\} \cup \Delta_2$$

where $Locs(\Delta_i) = \{l \mid \langle l, v \rangle \in \Delta_i\}$ or the set of all locations which would be affected by application of the update set i .

It is important to note that if either Δ_1 or Δ_2 are inconsistent, an inconsistent update set is returned. The definition of update set composition is then extended to the following definition [15, Def. 4.1.1]:

$$\Delta_{seq} = \Delta_1 \oplus \Delta_2 = \begin{cases} \{(l, v) \in \Delta_1 \mid l \notin Locs(\Delta_2)\} \cup \Delta_2, & \text{if } consistent(\Delta_1) \\ \Delta_1, & \text{otherwise.} \end{cases}$$

2.2.1 Iterative Composition

Simple iterative execution of a rule is provided by the **iterate**-rule

iterate
 R

The rule R will be executed until a termination condition exists: *successful* termination occurs when the update set becomes empty, and *failed* termination occurs when an inconsistent update set is produced by an iteration. Given that n iterations occur before termination of the rule and that $R^0 = \mathbf{skip}$, the n -th iteration of the rule can be described using **seq**

$$R^n = R^{n-1} \mathbf{seq} R$$

with the corresponding update set produced.

The **while**-rule, an extension to **iterate**, additionally allows for *conditional* termination, occurring when the guard g is satisfied:

while g
 R

2.3 Distributed Abstract State Machines

Distributed ASMs, useful for the design and analysis of distributed systems, extend Basic and Turbo ASM concepts with autonomously operating *agents*, each running their own program. If ASM M is a DASM, vocabulary Υ includes a finite universe $AGENT$ of agents, a unary function *program*, and a nullary function *self*. A new agent can be introduced into the DASM at any time by extending the domain $AGENT$.

The unary function *program* holds the program associated with a given agent. Behaviour of any agent a in T_i is defined by $program(a)$ as it is interpreted in S_i . If the value of $program(a)$ is an element representing rule R , the rule R will be executed whenever a is run.

To terminate a , $program(a)$ must be set to $undef$. Hence in a given step, the set of agents which may execute is defined as:

$$agentSet \equiv \{x \in AGENT \mid program(x) \neq undef\}$$

When the program of an agent a is executed, $self$ evaluates to a (i.e. $self = a$ for all references to $self$ in $program(a)$). The $self$ function allows an agent to store information relevant to itself. For example, a unary function $status$ could be used to hold status information for each agent (e.g. $status(self) := idle$ could mean that agent a is going into an idle mode).

DASMs can be further divided into *synchronous* (see [15, Chapter 5]) and *asynchronous* (see [15, Chapter 6]) varieties, each differentiated by the execution of their agents. A synchronous DASM is characterized by having a set of agents which execute their own programs in parallel, synchronized using an implicit global system clock. Essentially, at each step of M , all $a \in agentSet$ execute $program(a)$.

In contrast, with asynchronous DASMs, at each tick of the global clock a subset of agents are chosen for execution from the set of executable agents $agentsSet$. So at each step of M , the programs of all agents $a \in selectedAgentsSet$, where $selectedAgentsSet \subseteq agentSet$, are executed in parallel.

DASMs facilitate both the true-to-life modelling of mutually exclusive tasks performed by individual agents, and the analysis of interaction between agents via global state.

The reader should note that asynchronous DASMs are a superset of synchronous DASMs; notice the case when the algorithm used for agents selection during a step of an asynchronous DASM always yields $selectedAgentsSet \equiv agentSet$. We note that a single agent DASM is essentially a Basic ASM.

2.4 Additional Rule Forms

So called “syntactic sugar” is welcome in any language to facilitate ease of use and intuitive understanding. Quite a few additional rules have been introduced to ASMs over time for this purpose, and those used in this document will be briefly described here.

2.4.1 Temporary Variables

When performing calculations, temporary variables that are only necessary during computation of a result, may be useful. This helpful functionality is made available to the ASM modeller via the Turbo ASM `local`-clause which may be used with `named`-rules:

```
NamedRule  $\equiv$ 
  local  $f := t$ 
   $R[f]$ 
```

Here upon execution of `NamedRule`, the temporary function f is first initialized to the value of term t and then followed by execution of the body R . A call to this `named`-rule produces an update set as expected, however all updates to f are discarded. Thus, f can be used with impunity, resulting in no updates to it in the global state.

2.4.2 Return Values

In programming languages it is often useful for a subroutine to return a value to the caller. Similarly, we introduce a rule denoted by $l \leftarrow$ and defined in [15, Def. 4.1.7] (which we refer to as the `result`-rule), that provides such a mechanism allowing one to retrieve the intended return value of a named rule from a location determined by the `result`-rule call:

```
 $l \leftarrow$  NamedRule
```

Evaluation of this named rule would yield a set of updates as usual, however a side effect of this is that a special result would be assigned to the location l ; an update of the form $\langle l, \mathbf{result} \rangle$ is produced by the evaluation this rule. The named rule called is expected to have `result` (in this case both a keyword and function) as a free variable:

```
NamedRule  $\equiv$ 
   $R[\mathbf{result}]$ 
```

and to assign some value to this function during its execution.

While this method of retrieving returned values is sufficient in many cases, it is not always practical. Its syntactic form only allows for the return of a value into a given variable, and so it cannot be used inside a complex expression. For example, one has to write

```

par
   $x_1 \leftarrow R_1$ 
   $x_2 \leftarrow R_2$ 
seq
   $x := x_1 + x_2$ 

```

instead of the more natural

```

 $x := R_1 + R_2$ 

```

The **return**-clause is introduced in [23] to remedy just this problem

```

NamedRule  $\equiv$ 
  return  $x$  in
     $R[x]$ 

```

When a named rule with a **return**-clause is found in an expression, its rule body R is evaluated, and the final value of x resulting from its execution is substituted into the expression where it is called. Since it is semantically atypical for an expression to produce an update set, the update set produced by such a named rule is discarded.

2.4.3 Substitution

Frequently, a simple value is required to specify some aspect of a model, but the calculations done to derive said value are quite tedious. Other times the result of a term is required many times in a specification, but the process of deriving the result is not important for the understanding of behaviour. In Section 2.2 for example, we used the unary function *Locs* in the definition of sequential composition, giving its meaning after. In such cases it is convenient to separate the use of a result, from its derivation.

The **let**-rule form provides this functionality

```

let  $x = t$  in
   $R[x]$ 

```

where the term t is evaluated and assigned to the variable x just before evaluating R . Named rules may also be equipped with a **where**-clause

NamedRule \equiv
 $R[x_1, \dots, x_n]$
 where
 $x_1 = t_1$
 \vdots
 $x_n = t_n$

Here x_i is assigned the value resulting from the evaluation of term t_i just before the evaluation of every instance of x_i in R . It is good practice when using these facilities in a model to give appropriate names to the free variables to allow for correct interpretation (by the reader) of the value stored in the variable without needing to refer to the calculations which produced it.

2.5 Control State ASMs

The flow of control of an application at a high level of abstraction is often better understood when depicted in pictorial or chart form. Here we introduce *control state ASMs*, which allow for the description of control flow using a diagrams. A control state ASM is an ASM whose rules may be, at a high level of abstraction, defined pictorially as depicted in Figures 2.2-2.5. Similar to the traditional flow chart, conditions are depicted using the typical diamond type symbol (\diamond), so called *control states* are depicted using circular symbols (\circ), rules to be executed are depicted as rectangular symbols (\square), and flow depicted with directed arrows (\rightarrow).

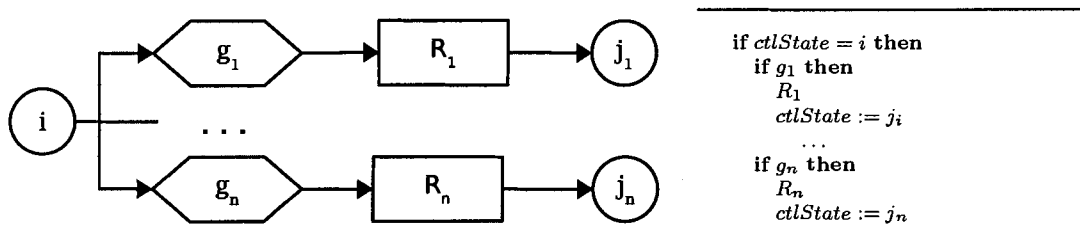


Figure 2.2: Control State ASM : Conditional execution of rule with conditional control state change.

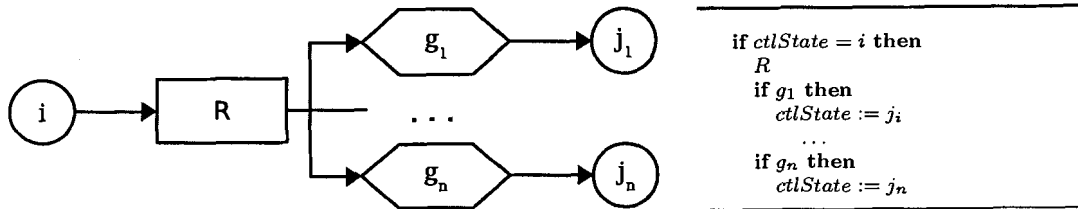


Figure 2.3: Control State ASM : Execution of single rule with conditional control state change.

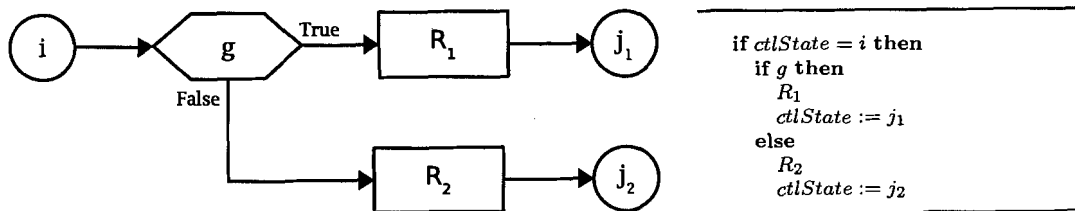


Figure 2.4: Control State ASM : Execution rule and control state change based on truth of single condition.

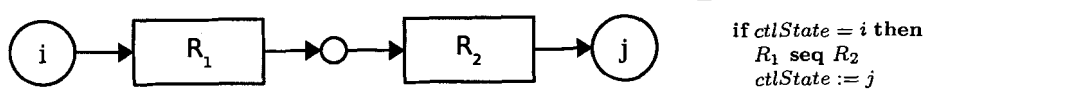


Figure 2.5: Control State ASM : Sequential execution of rules.

The control states² $ctlState \in \{1, \dots, m\}$ resemble the internal states of finite state machines, and can be used to describe different system modes (see [15, Section 2.2.6]). It is important to notice that any transition between control states represents a step of the underlying machine.

The examples given in Figures 2.2-2.5 illustrate common control state ASM forms, and the corresponding ASM rule which describes their behaviour; basic conditional execution and state change along with sequential execution are shown. While we do not give examples here, it should be clear that iterative execution, parallelism, and non-deterministic choice can all be easily described with control state ASMs as well.

2.6 Background Classes

In some applications, it may be useful to represent and make use of well defined data structures as elements of the state, thus increasing the working space with such elements. For example we may wish to use binary trees in our model. Recall (from Section 2.1) that fresh elements can be allocated for use by an algorithm, but only from the reserve; all members of the reserve set exist in all states of an ASM M , and so new elements can not actually be added to any state S of M at any time. Adhering to this foundational principle of ASMs, one is required to define each binary tree used, by first using the **import**-rule to retrieve a new element from the reserve and specifying its contents, ultimately worrying about the nature of these newly created elements.

However, it would be much more convenient to assume that every state contains all *hereditarily finite* binary trees³ over all its *atomic* elements⁴. This is achieved by assuming that the reserve, which has no predefined internal structure over it, has some external structure over it. Thus, when a reserve element is imported, binary trees containing it already exist and do not need to be created separately (by importing additional elements and appropriately defining the membership relation on them). Any binary tree could then just be used by an algorithm, with the knowledge that it has already been defined and ready for use. This idea can be trivially extended to other sorts of data structures, such as sets,

²Control states are not states of the underlying ASM, but rather are modes of the system being modelled.

³The set of all hereditarily finite binary trees contains finite binary trees of elements (and finite binary trees of these, etc.).

⁴Here an *atomic* element refers to any element of the state which is not a binary tree.

maps, strings, numbers, and the list goes on.

When the assumption is made that all states of an ASM include all elements of a particular sort, it is said that the ASM uses the *background class* of that sort. The concept of background classes was formalized in [4].

2.7 Partial Updates

Basic ASM updates provide for the replacement of the value v at a given location l with a new value v' . However, it is often convenient to look at a value as a complex data structure which has its own internal state, that may itself receive multiple incremental updates during a step, resulting in a total update of its internal state. The narrow view of the modification of state in Basic ASMs precludes the ability to make such *partial updates*.

For example, during the process of modelling a message passing protocol, it might be convenient to keep messages in a set. Let us assume that *messages* is a set which currently contains some messages. If Basic ASM updates were all that were available, and we wish to add messages msg_x and msg_y to *messages*, but potentially independently of each other, we may attempt to accomplish this like so:

```

if  $g_x$  then
   $messages := messages \cup \{msg_x\}$ 
if  $g_y$  then
   $messages := messages \cup \{msg_y\}$ 

```

Notice that our attempt here to add messages msg_x and msg_y into the set *messages* can succeed only if one of the messages is added in a single step. In the case that both guards are satisfied, the updates produced by attempting to add msg_x and msg_y simultaneously would result in an inconsistent update set. Therefore the above approach is not sufficient. Using partial updates solves this problem:

```

if  $g_x$  then
  add  $msg_x$  to  $messages$ 
if  $g_y$  then
  add  $msg_y$  to  $messages$ 

```

Essentially, individual partial updates of a step are first collected and then *integrated* (see [38]) into a total update, that update resulting from the total change to *messages* caused

by all the partial updates.

Partial updates have been explored in [38] and [39]. These papers introduce a general mathematical framework for handling these incremental changes in ASMs.

2.7.1 General Mathematical Framework

The goal of [38] and [39] was to allow for the incremental change of elements, while preserving the traditional ASM setting, namely that location contents are changed by only updates.

The authors introduce the notion of *particles*, which represent total and incremental changes of an element; particles are, in essence, mathematical functions representing the particular change to be made on an element of U . For example, a regular update to a value v' can be represented by a particle `overwritev'`. Similarly the update resulting in the addition of element v'' to a set may be represented by a particle `setaddv''`.

Partial updates, which are in this framework produced by both **update**-rules and **partial update** type rules (such as the **add-to**-rule introduced in our message example), consist of a location and a particle: $pu = \langle l, p \rangle$.

Once all partial updates produced are collected into a *partial update multiset*,⁵ (denoted $\tilde{\Delta}$, notice the tilde), an update set of total updates Δ is produced by the integration of all partial updates for each individual location. The binary operator \circ results in one particle, being the combination of two particles in order. A total update for any location can only be produced by integration, if particles can be combined in an order independent manner. So if the particles `setaddmsgx` and `setaddmsgy` were produced by our example, integration could only produce a total update for *messages* successfully if:

$$\text{setadd}_{\text{msg}_x} \circ \text{setadd}_{\text{msg}_y} \equiv \text{setadd}_{\text{msg}_y} \circ \text{setadd}_{\text{msg}_x}$$

A new notion of consistency is introduced for a partial update multisets: $\tilde{\Delta}$ can only be considered *consistent* if for every location, all partial update particles can be combined in an order independent manner as explained above.

⁵The authors of [38, 39] use a multiset to collect partial updates, as they have found that integration may depend on the multiplicity of partial updates; for example multiple increments to integer counters [38, Section 6].

2.7.2 Turbo ASMs and Sequential Composition

In regards to Turbo ASMs, we note that the sequential composition of partial updates should result in partial updates rather than total updates; put in another way, partial updates should not be integrated by a Turbo ASM rule.

Recall the **seq**-rule:

$$R_1 \text{ seq } R_2$$

With the introduction of partial updates, R_1 and R_2 result in partial update multisets $\tilde{\Delta}_1$ and $\tilde{\Delta}_2$ respectively, and the **seq**-rule itself should yield a multiset of partial updates $\tilde{\Delta}_{seq}$ resulting from their composition. Partial update multisets composed sequentially (via $\tilde{\oplus}$) are defined as follows

$$\tilde{\Delta}_{seq} = \tilde{\Delta}_1 \tilde{\oplus} \tilde{\Delta}_2 = \begin{cases} \{\langle l, p_2^l \circ p_1^l \rangle \mid l \in L\}, & \text{if } \text{consistent}(\tilde{\Delta}_1) \\ \tilde{\Delta}_1, & \text{otherwise.} \end{cases}$$

where $L = \text{Locs}(\tilde{\Delta}_1) \cup \text{Locs}(\tilde{\Delta}_2)$, and p_i^l refers to all particles operating on location l in partial update-multiset $\tilde{\Delta}_i$. Notice that particles are combined⁶ such that $\tilde{\Delta}_1$ occurs before $\tilde{\Delta}_2$; in a **seq**-rule, relative ordering of particles between the two partial update-multisets must be preserved.

2.7.3 Rule Forms

In this document we use one of two partial update rule forms which operate on both sets and multisets: the **add-to**-rule and the **remove-from**-rule. The **add-to**-rule causes the addition of the element v_e , resulting from the evaluation of term t_e , into the set or multiset v_s , resulting from the evaluation of term t_s :

$$\text{add } t_e \text{ to } t_s$$

In contrast, **remove-from**-rule results in the removal of element v_e from the set or multiset v_s :

⁶We have $p_2^l \circ p_1^l$ rather than $p_1^l \circ p_2^l$ because of the way particle combination works: $(p_2^l \circ p_1^l)(x) = p_2^l(p_1^l(x))$ [38, Remark 8.2].

remove t_e from t_s

Note that if v_e does not exist in v_s this rule has no effect. Also note that when v_s is a multiset that currently contains multiple instances of v_e , only one instance of the element is removed from the multiset.

2.8 Modelling Computer Languages

ASMs have been used to formalize many computer languages (see Section 4). We note that another widely accepted methodology used to formalize the semantics of computer languages is *denotational semantics* [49]. Here we briefly discuss the ASM paradigm and denotational semantics in the context of computer language development, and show why the ASM method is more appropriate for our purposes. For more in-depth coverage of this topic, we refer the reader to [8, 20].

The denotational semantics methodology is used to define language semantics mathematically, modelling data types as domains (i.e. sets categorizing data) and modelling programs of a language as functions between domains, thus allowing for the precise definition of certain language semantics. However denotational semantics abstracts away from the dynamics of computation – change over time – and thus cannot easily capture all semantics. So, while denotational semantics are powerful and concise in certain cases, limitations in what can be expressed with it make it appropriate only for certain language constructs as mentioned in [8]:

Such [denotational] semantics . . . allows one to establish many useful properties, but there is a price. Not all programming constructs lend themselves to such treatment which limits the applicability of the method.

The *operational semantics* [49] approach defines the operational semantics of a program in a language via sequence of internal machine configurations that a machine moves through over time as it executes the program instructions, thus capturing the dynamics of computation. A common way to rigourously formalize the operational semantics is through state-transition systems [46], and all ASMs are state-transition systems. ASMs with their abstract state composed of interpretations of functions between elements of universes, allow

for a precise, mathematically-structured definition of semantics like denotational semantics, while also incorporating the notion of state-transition. With ASMs one can also define operational semantics at the desired level of abstraction, avoiding unneeded complexity in formalization.[8]. The suitability of the ASM method for formalization of computer programs is stated concisely in [20]:

A bitter consequence is that applications of such domain based methods [denotational semantics] are usually restricted to relatively simple properties for small classes of programs; ... pure versions of various functional programs (pure instead of common LISP programs), Horn clauses or slight extensions thereof instead of Prolog programs, structured WHILE programs instead of imperative programs appearing in practice (for example Java programs with not at all harmful, restricted forms of go to). ... The add on of ASMs with respect to denotational methods is that properties and proofs can be phrased in terms of abstract runs, thus providing a mathematical framework for analyzing also runtime properties, e.g. the initialization of programs, optimizations, communication and concurrency (in particular scheduling) aspects, conditions which are imposed by implementation needs (for example concerning resource bounds), exception handling, etc.

Essentially the ASM paradigm should be used where the expression and analysis of operational behaviour is important. Because we wish to formalize and validate both CoreASM engine operation and language semantics, the ASM method is appropriate for our application. It is interesting to note, however, that Basic ASMs have been defined using denotational semantics [40].

2.9 Notational Conventions

All ASM specifications presented in this thesis use the following notational conventions for improved readability:

- Named rules and always begin with an upper case letter. When a name contains multiple words, individual words begin with capital letters, and are not separated by any alternate characters. When called from within another rule, the name will appear in sans serif type (e.g. NamedRule)

- When a named rule or submachine is first defined, their name appears in bold-face sans serif type followed by the equivalence symbol, and their rule body R (e.g. **NamedRule** $\equiv R$)
- Domains appear in all capital letters, with larger sized capitals beginning individual words (e.g. DOMAINNAME).
- Function and predicate names always begin with a lower case letter and appear italicized. When a function or predicate name contains multiple words, individual words after the first begin with capital letters, and words are not separated by any alternate characters (e.g. *functionName*).
- All ASM reserved words appear in boldface lowercase letters (e.g. **skip**).
- Comments are always prefixed with double slashes (e.g. // Comment).
- ASM specifications can found throughout this document, intermingled with text. Such specification segments are separated from the text by two horizontal lines: A thick line (——) indicating the start of the segment and a thin line (——) indicating its end. Some segments may contain a title at the top right hand corner of the segment, appearing in small sans serif type (e.g. ASM Segment Title):

ASM Segment

// Body

- ASM specifications which are numbered for reference elsewhere are enclosed in a full frame, with their number and caption below:

ASM Segment
// Body

Specification 2.1: The specification description.

Such specifications also appear in the List of Specifications on Page xiv of this document.

“A compiler’s primary function is to compile, organize the compilation, and go right back to compiling. It compiles basically only those things that require to be compiled, ignoring things that should not be compiled. The main way a compiler compiles, is to compile the things to be compiled until the compilation is complete.”

— *Student’s wrong answer on CS exam.*

Chapter 3

Compiler Theory Preliminaries

In this chapter we give a brief introduction of compiler theory concepts necessary for an understanding of the CoreASM project and this thesis. We begin with a short comparison of interpretation and compilation in the context of computing. We follow this with an introduction to language definition, input representation and evaluation, and operator specific concepts. We end with a discussion of types and semantic analysis in computer languages. All information presented can be found in [1] unless otherwise noted.

3.1 Interpreter vs. Compiler

An *interpreter* interprets a computer program, resulting in its execution. In contrast a *compiler* simply translates a computer program from its language to another, and involves no execution of any kind.

However, both an interpreter and a compiler do share some components, namely the *lexical analyzer*, *syntactic analyzer* (parser), and *semantic analyzer*. Each component, respectively, represents a successive stage in the processing of the input. While we restrict our discussion to the case where their execution (in the error-free case) results in an intermediate representation of the input program, note that some approaches to compilation translate input directly to output, and thus do not produce an intermediate representation.

The purpose of the lexical analysis phase is to break up the input into small portions, called *tokens*; tokens are found by pattern matching against input, looking for specific segments of text (e.g. reserved words, numbers, identifiers). Tokens are fundamental units of an input language.

A syntax analyzer takes tokens from the lexical analysis phase and creates a representation of the input, as a tree. The tree is built based on syntactic constraints of the input language, which have been encoded in its *context-free grammar* (grammar for short); in essence, a grammar gives a precise syntactic specification of the input language. Syntax analysis ensures that input is correctly structured by making certain that the order in which tokens were found is acceptable.

During the semantic analysis phase, checks are made to ensure that the meaning of the input is unambiguous and the input program instructions are supported by the computer language. One such check is *type checking*¹, where (amongst other things) checks are made so that, for instance, all operators have operands of a *type* (e.g. integer, real, string) that they are capable of operating on.

From this stage onward in their respective processes, an interpreter will use its representation of the input to interpret the meaning and execute the tasks which the input language describes, whereas the compiler will use the representation to translate the meaning into the target language.

3.2 Language Definition

The syntax of a language is specified by a *grammar*, which naturally describes the hierarchical construction of the language it depicts. A grammar consists of four components²:

1. A set of *terminal* symbols, each terminal being associated with a particular token³.
2. A set of *nonterminals*.
3. A set of *productions* where each production consists of a nonterminal called the left-hand side (LHS) of the production, an arrow, and a sequence of tokens and or nonterminals, called the right-hand side (RHS) of the production.
4. A single nonterminal, designated as the *start* symbol.

¹Many other semantic error checks can be performed, including those for variable and subroutine declaration, variable initialization, etc. However, we focus only on type checking here because it alone is relevant to this work.

²These four points were taken from [1] with little modification.

³Note that different character strings in input, called *lexemes*, may be represented by a single token (e.g. “foo” and “bar” could both be represented by the Identifier token).

When presenting grammars or grammar segments, we follow these conventions:

- Typewriter font will always be used (e.g. `typewriter font`).
- Nonterminals will always begin with an upper-case letter. When a nonterminal contains multiple words, individual words begin with capital letters, and are not separated by any characters (e.g. `NonterminalExample`).
- Terminal symbols will always be delimited by single quotes (e.g. `'terminal'`).
- Arrows will be depicted as `: ->`.
- If an entire grammar is being depicted, the designated start symbol will be the LHS of the first production listed. Otherwise, the grammar segment will begin with ellipses.
- Because more than one production can have the same LHS, we allow a LHS to have multiple RHSs for convenience. Multiple RHSs of a production are separated with a *pipe* (e.g. `|`).

We will consider for the purposes of illustration, a simple language which contains arithmetic expressions consisting of single digits along with multiplication and minus signs. The grammar for this language is shown in Figure 3.1:

```
Expr -> Expr '-' Digit | Expr '*' Digit | Digit
Digit -> '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

Figure 3.1: A simple expression grammar.

The grammar presented in Figure 3.1 will accept strings `1 * 5 - 7` or `2 * 3 - 8`, but not `1 * 10` or `-5`.

3.3 Input Representation and Evaluation

Because the process of syntactic analysis (or parsing) is complicated and technical, we do not discuss it in detail here. However it suffices to say that the parser uses the grammar to build a representation of input as a tree of tokens. This tree may be a *parse tree*, which is an exact representation of the nonterminals and terminals used for the parse, or it may be an

abstract syntax tree (AST), which abstracts away from the exact symbols of the grammar used but still describes the meaning of the input. Figures 3.2 and 3.3 show the parse tree and AST for the expression $2 * 3 - 4$; notice that the AST is much smaller than the parse tree. ASTs are generally more practical representations of input than parse trees are.

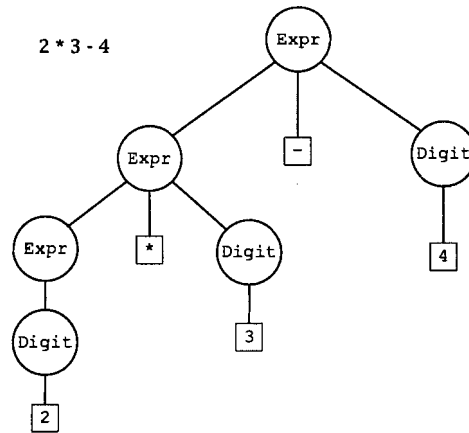


Figure 3.2: Example parse tree.

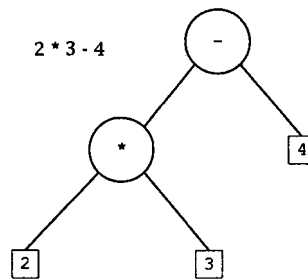


Figure 3.3: Example abstract syntax tree.

To determine the meaning of any tree representation, a tree is traversed in a depth-first *postorder*⁴ fashion; all nodes are visited after their children⁵. In a compiler, visiting a node

⁴Here we are describing the general case. However, in an interpreter for example, during interpretation of control structures such as *loops*, children may be revisited after visiting the parent.

⁵The order in which the children of a node are visited (if they are visited at all) is dependent on the node itself.

entails translation of program instructions, while in an interpreter, visiting a node results in the execution of program instructions. However, it is important to note that the end result of executing a compiled program and interpreting the same program is identical. In our AST examples we shall focus on this end result through interpretation, and so we use the term *evaluated* to refer to a node that has been visited.

The results of evaluated subtrees are used by the parent node. This is illustrated in Figure 3.4, where, by first evaluating the its subtrees, and then the root of the AST, a result of 2 is derived for $2 * 3 - 4$.

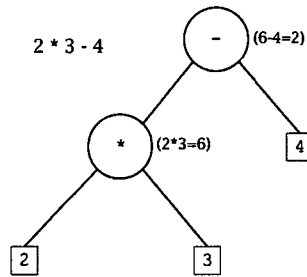


Figure 3.4: Evaluated abstract syntax tree.

3.4 Operator Classification and Precedence

For our purposes, we define an *operator* to be one or more symbols which denote a process or action called an *operation*, which takes one or more arguments called *operands*; the evaluation of an operator results in a value. Operators can be divided into classes based on their syntactical form: *unary* (e.g. numerical negation “-”), *binary* (e.g. multiplication “*”), *ternary* (e.g. conditional assignment “?:”), *grouping* (e.g. expression grouping “()”) and *indexing* (e.g. array index “[]”).

Every operator can also be classified as *left associative* (LA) or *right associative* (RA). *Associativity* of operators is defined by the side of an expression from which evaluation begins: LA operators are evaluated from left to right; RA operators are evaluated from right to left. We shall use the binary operators for arithmetic subtraction (“-”) and assignment (“=”), which are LA and RA respectively, as an example: notice that $a - b - 4$ is equivalent to $(a - b) - 4$ whereas $a = b = 4$ is equivalent to $a = (b = 4)$. Recall that

evaluation of input requires postorder traversal of the AST. As such the representation for two seemingly similarly constructed expressions $a - b - 4$ and $a = b = 4$, are two completely different ASTs (Figure 3.5).

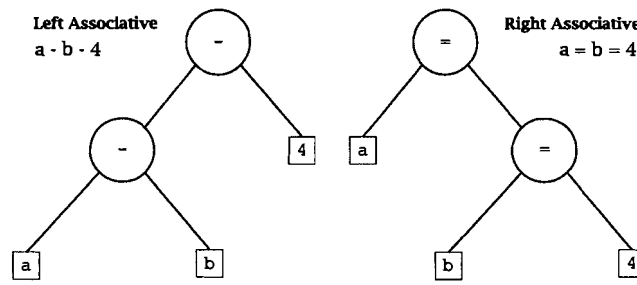


Figure 3.5: AST for left and right associative operators.

While associativity defines how multiple instances of the same operator are to be evaluated, *operator precedence* (OP) defines how multiple operators are to be evaluated in the absence of parenthesis. According to arithmetic conventions the expression $5 - 6 * 8$ is equivalent to $5 - (6 * 8)$ and not $(5 - 6) * 8$; arithmetic multiplication has a higher precedence than does arithmetic subtraction.

Any grammar can be structured such that the parse tree (and hence the AST) produced by a parse, has OP in mind. A grammar in which the acceptable order of operations is encoded is called an *operator precedence grammar* (OPG). In an AST produced by such a grammar, higher precedence operators appear lower in the tree, where they will be evaluated before operators of lower precedence (see Figure 3.6).

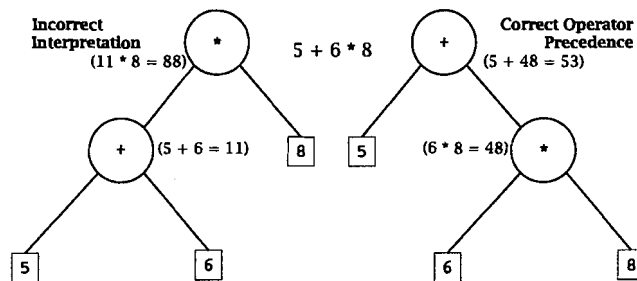


Figure 3.6: AST resulting from OPG.

The OPG presented in Figure 3.7 results from simple modifications to the grammar shown in Figure 3.1. It implements the correct operator precedence for arithmetic multiplication and subtraction, ensuring that multiplication will be evaluated before addition:

```
Expr -> Expr '-' Term | Term
Term -> Term '*' Digit | Digit
Digit -> '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

Figure 3.7: Expression grammar, now an OPG, taking into account operator precedence.

3.5 Types and Semantic Analysis

In programming languages, variables, and hence memory locations, hold a certain *type* of value. In examples to follow, we assume that the variable “*a*” holds values of an integer type called *Integer*, the variable “*b*” holds values of a real type called *Real*, and the variable “*c*” holds values of a string type called *String*.

Type checking ensures that only operands of a certain type can be used with a given operator. Given an arithmetic operator for addition (“+”), a type checker would prevent $a + c$ from being accepted, as arithmetic addition is undefined for an Integer and a String.

An “overloaded” symbol is one that has different meanings depending on its context [1]. Operators may be *overloaded*⁶ by providing an implementation for operands of different types, with different implementations provided by either the user or the language itself. For example, the “+” operator may be overloaded for use with Integer addition or Real addition. However, for it to work with operands of different types, the Integer must be converted to a Real or vice versa. The compiler or interpreter is generally aware of all types of operands provided by the language, and which conversions are possible.

The process of automatic conversion from one type to another is called *coercion*. So, if $a = 1$ and $b = 2.2$ then before $a + b$ can be computed, a must be coerced to the value 1.0, resulting in the evaluation:

$$a + b = 1.0 + 2.2 = 3.2$$

⁶Note that a *polymorphic* operator is not the same as an overloaded operator; the former has a single implementation for all types, while the latter has different implementations for different types.

Programming languages may also provide facilities for the programmer to *explicitly convert* information from one type to another. This type of conversion is also known as *casting* [53].

3.5.1 Dynamic vs. Static Typing

A *statically typed* (or *strictly typed*) computer language requires the user to specify the type of every variable used, whereas *dynamically typed* languages do not have this restriction. In dynamically typed languages, a single variable may, during execution of a program, hold information of multiple types.

Thus, type checking for languages such as C++ and Java, which use static typing, is done at compile time. Dynamically typed languages such as Perl, Python, and PHP do their type checking at interpret-time.

3.5.2 Strong vs. Weak Typing

As mentioned earlier, the compiler or interpreter is aware of all type conversions supported by a language, if any. The coercions which are supported⁷ allow for variables of one type to be automatically viewed and used as variables of other types where it is appropriate (e.g. where an operand is of one type but the operator works only with another type). A language that places many restrictions on how variable types can be viewed and used is said to be *strongly typed*, while a language that has few such restrictions is said to be *weakly typed* [53].

For instance, a strongly typed language may only allow numeric types to be coerced into other numeric types. Recalling our earlier example where $a=1$ and $b=2.2$, a strongly typed language would be able to compute $a+b$ with a simple coercion of a to its equivalent Real value. The languages C++, Java, and Python are all strongly typed languages.

In contrast, a weakly typed language may also allow for non-numeric types to be viewed as numbers. For example if $a=5$ and c holds the String "3.3", a weakly typed language would, before the evaluation of $a + c$, coerce a to 5.0 and c to 3.3, resulting in:

$$a + c = 5.0 + 3.3 = 8.3$$

⁷To simplify the discussion we focus on automatic conversions that involve actual coercion of the source value to the target type, versus the simple reinterpretation of information in memory as the target type, as is done in weakly typed languages like C.

To achieve this same outcome in a strongly typed language, explicit conversion of the String "3.3" would be required prior to expression evaluation. Both PHP and Perl are weakly typed languages.

“Computer language design is just like a stroll in the park. Jurassic Park, that is.”

— Larry Wall

Chapter 4

CoreASM Overview

Semantic foundations of many widely known computer languages have been modelled using the ASM formalism. These include industrial system design languages like the ITU-T standard for SDL [32, 47, 22, 41]; the IEEE language VHDL [12, 11] and its successor SystemC [45]; programming languages like JAVA [51, 21], C# [10] and Prolog [6, 7]; and the Business Process Execution Language for Web services [29, 30, 28]. As such, the ASM formalism is a natural choice for the formal specification of the CoreASM engine, language, and toolset.

In this chapter we shall give a basic overview of the CoreASM engine and language. First we provide a high-level architectural view of the CoreASM engine. Then we present its components in some detail and discuss the extensibility provisions in its architecture. An abstract specification of the CoreASM language is then presented, along with several examples of how the core language and its extensions are specified.

The CoreASM tools presented in this chapter are the result of incremental design and specification presented by Farahbod *et. al* in [25, 24, 23, 26]; various diagrams and the notation used, have been borrowed and/or adapted from these earlier and upcoming documents. We introduce the engine with the intention of refining the description of certain components' behaviour and making amendments to this design, in subsequent chapters.

Because the specification paradigm being modelled is identical to that being used to model it, at times it may not be clear which ASM we are referring to. In such cases, we refer to the CoreASM engine and its execution as the *simulated machine*, and the underlying ASM used to model it as the *underlying machine*.

4.1 Engine Architecture

The CoreASM engine is composed of four modules, namely the *Parser*¹, the *Interpreter*, the *Scheduler*, and the *Abstract Storage* (Figure 4.1²). All these components work together to simulate an asynchronous DASM run. The *Control API*, a liaison between the engine and environment, facilitates and coordinates their interaction.

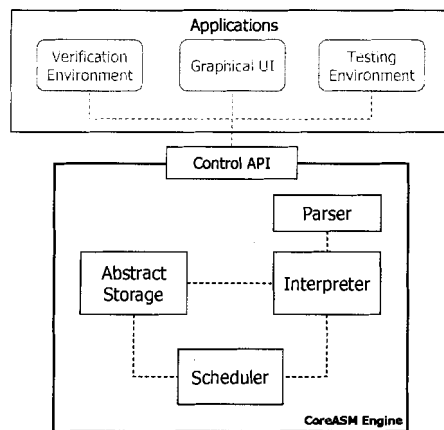


Figure 4.1: Architectural Overview of CoreASM Engine

First the Parser produces an AST representing the CoreASM specification provided. The Interpreter then traverses the AST, executing rules and evaluating expressions, and then it collects all updates produced. It is the responsibility of the Abstract Storage to manage the data model, and in particular the basic function and predicate locations of simulated machine state. The current state of the simulated machine is stored along with a history of previous states of the current run. The history can be used to explore the series of moves resulting in the current state or to rollback to a previous state and reengage the computation from there.³ While evaluating a program, the Interpreter interacts with the Abstract Storage in order to obtain the values of locations in the current state. As there may be multiple agents which execute in a single DASM step, the Scheduler is charged with selecting a set of agents that will contribute to the next computation step and with coordinating the

¹The Parser module handles both lexical and syntactic analysis of specifications.

²This figure was reproduced from [24] by permission.

³While the rollback mechanism is able to rollback the machine state, it is important to mention that it has no control over the environment.

execution of those agents. The Scheduler must also handle cases of inconsistency in update sets generated in each step. Essentially the Scheduler coordinates the execution of each step of the simulated machine.

The actions which constitute a single step in the CoreASM engine are as follows (refer also to Figures 4.6-4.9):

1. The Control API sends a *STEP* command to the Scheduler.
2. The Scheduler retrieves the entire set of executable agents from the abstract storage.
3. The Scheduler selects a subset of these agents to perform computation during the next step.
4. The Scheduler selects a single agent from the subset, assigning it to the special variable *self* in the Abstract Storage.
5. The Scheduler instructs the Interpreter to run the program of the current agent (which can be retrieved by evaluating *program(self)* in the current state).
6. The Interpreter executes the program.⁴
7. When evaluation is complete, the Interpreter notifies the Scheduler that the interpretation of the agent's program has completed.
8. The Scheduler chooses a different unevaluated agent from the subset. If there are no more unevaluated agents left in the subset, the Scheduler calls the Abstract Storage to fire the accumulated updates.
9. The Abstract Storage notifies the Scheduler whether the update set has any conflicts or it was successfully fired. An inconsistent update set will lead to selection of a previously untested subset of agents (if any) and their execution in this step, while a successful firing of the update set will cause control to be sent back to the Control API.

⁴This generally involves interaction between the Interpreter and the Abstract Storage to evaluate terms against the current state.

4.1.1 CoreASM Components

In this section we present the modules of the CoreASM engine in greater detail, along with its extensibility mechanisms.

The engine is modularized along two dimensions (see Figure 4.2⁵). Until now we have presented the architecture in terms of the four main components of which it is composed: Parser, Interpreter, Scheduler, and Abstract Storage. However, the second dimension allows us to distinguish between what is absolutely necessary to DASM semantics, which is contained in the *kernel* of the engine, and what is not, which is handled by *plug-ins*. The language accepted by the CoreASM engine may be progressively extended using plug-ins, thereby augmenting the basic functionality of the kernel.

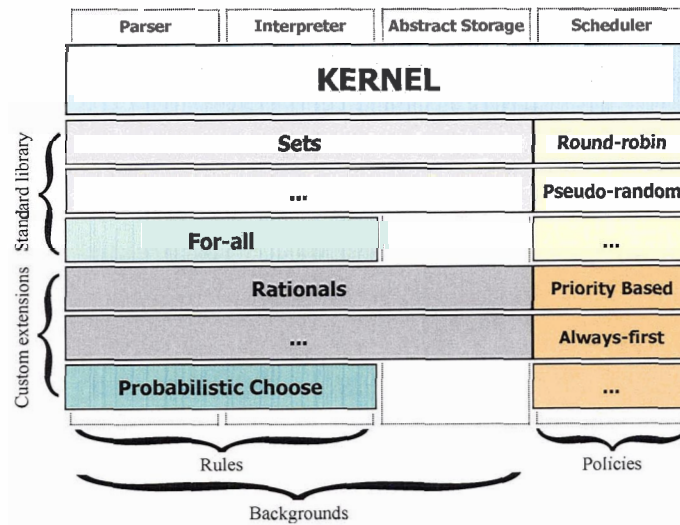


Figure 4.2: Layers and Modules of the CoreASM Engine

Before a CoreASM specification can be interpreted, the Parser must generate an *annotated* AST for it. Each node in the tree produced may be annotated with a reference to the plug-in where the corresponding syntax and semantics for its evaluation is defined. Our example in Figure 4.3⁶ has nodes associated with the Boolean, Set, and Number plug-ins; the Interpreter and Abstract Storage will use this information to correctly evaluate the nodes with respect to their corresponding plug-ins.

⁵This figure was reproduced from [24] by permission.

⁶This figure is based on one from [24].

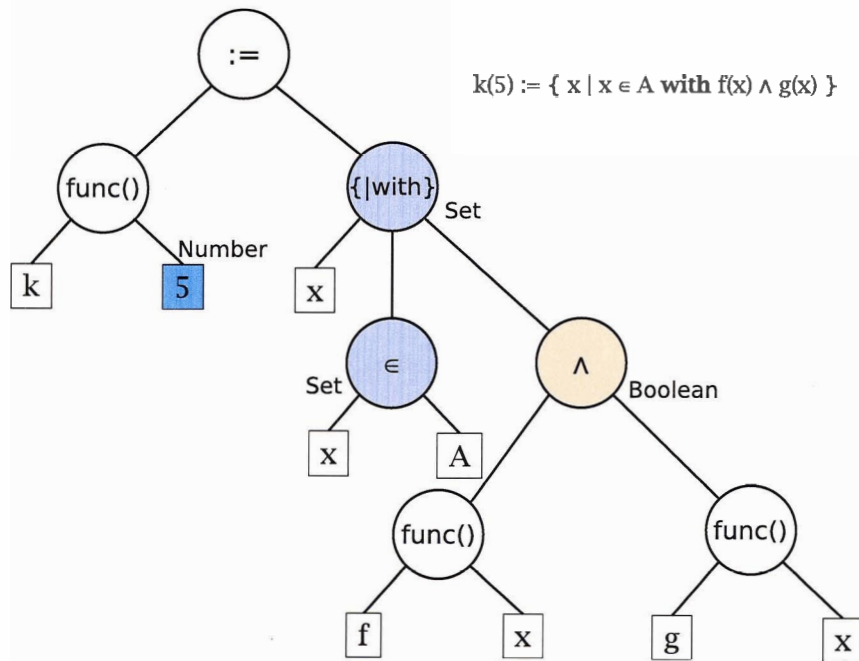


Figure 4.3: Sample annotated AST

The Interpreter traverses the AST, executing rules and potentially giving control to plug-ins when coming across nodes for which they are responsible; any node containing no plug-in information is handled directly by the kernel. While traversing the tree, a set of updates is generated by the evaluation of all rules. Interaction with Abstract Storage is required for the evaluation of terms against the current state.

The current state of the simulated machine is maintained by the Abstract Storage module. Interfaces provided by this module allow the retrieval of values from any location in the current state and for the application of consistent updates upon the completion of a successful step. In the underlying machine, simulated state is modelled as a map from locations to elements of the universe `ELEMENT`.



The Scheduler orchestrates the evaluation of each step in the DASM run. A step is initiated upon receipt of a `STEP` command from the Control API. First a subset of executable agents is chosen to participate in the next step, requiring interaction with the Abstract Storage to retrieve the current set of agents. For each agent in this subset, the Scheduler passes control to the Interpreter to evaluate the program of the agent, and when evaluation of the agent's program is complete, it collects the set of updates generated. When all selected

agents have been executed, all updates generated (if consistent) are then applied, resulting in the next state. The environment is notified of the final status of the step attempt via the Control API, thus completing the step.

4.1.2 Engine Life-cycle

Informally, the process of executing a CoreASM specification using the CoreASM engine is:

1. Initializing the engine
 - (a) Initializing the kernel
 - (b) Loading the plug-ins library catalogue
 - (c) Loading and activating plug-ins from a standard library
2. Loading a CoreASM specification
 - (a) Parsing the specification header
 - (b) Loading further needed plug-ins as declared in the header
 - (c) Parsing the specification body
 - (d) Initializing the Abstract Storage
 - (e) Setting up the initial state
3. Execution of the specification
 - (a) Execute a single step
 - (b) If termination condition not met, goto 3a to execute next step.

A high-level formal specification of this procedure is provided as a control state ASM, and is shown in Figures 4.4-4.9. During execution of a CoreASM specification, control moves between the different components of the engine. As an aid to the reader, modules and the Control API are depicted in one of five colours, and portions of the process handled by a given component are enclosed in an appropriately coloured and labelled rounded box (). When control moves out of one component and into another, the module to which control moves is written in a note shape (.

For the remainder of this section we will walk through the control state ASM model; note that lower level interfaces provided by the modules of the simulated machine, and used

in the model, are formally defined in Appendix A. The ASM rules and conditions making up a single step of the simulated machine (see 3a above) will be given special attention. The current control state of the model is stored in the variable *engineMode*; at times we refer to this as simply the *mode* of the engine. When the engine is first executed, it begins its life in the *Idle* mode in the Control API, waiting for one of three commands (i.e. *INIT*, *LOAD*, and *STEP*). As commands from the environment arrive, they are inserted into a queue and evaluated in FIFO order.

When given the *INIT* command, the engine starts the initialization process (Figure 4.4⁷), first initializing the kernel, then creating a catalog of all plug-ins currently available, and finally activating all plug-ins which are included in the *standard library*. Once initialization is complete, the engine again waits in *Idle* mode, ready for another command.

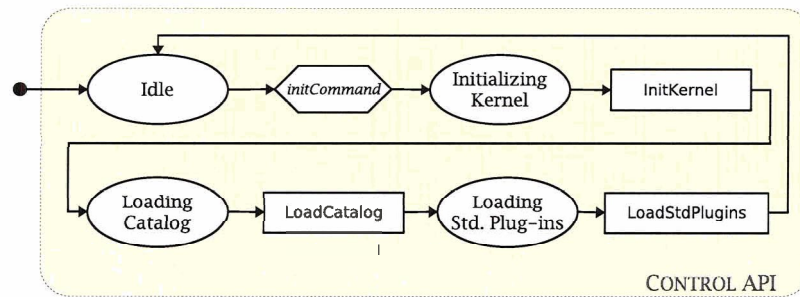


Figure 4.4: Control State ASM of Engine Initialization

The *LOAD* command causes a chain of events resulting in the loading of a specification (Figure 4.5⁸). The engine gives the CoreASM specification to the Parser which examines the header of the specification and determines which plug-ins are required for its execution. This information is used by the Control API to load the necessary plug-ins. Once the plug-ins (and thus their extensions to the engine) are loaded, the Parser is called to parse the specification, with the result being an AST. Based on this AST, the Abstract Storage will initialize the simulated machine state data structure with required functions, and the Scheduler will load the initial state of the simulated machine. Once the loading of the specification is complete, the engine again returns to *Idle* mode.

⁷This figure was reproduced from [26] by permission.

⁸This figure was reproduced from [26] by permission.

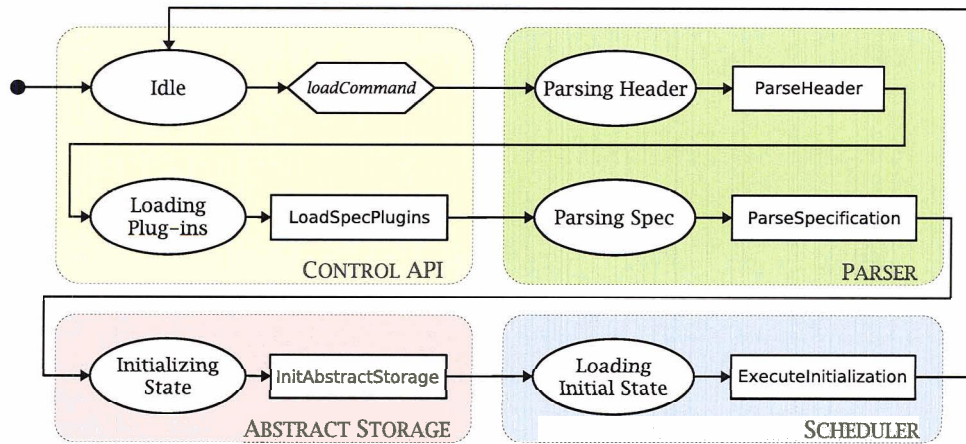


Figure 4.5: Control State ASM of Loading of CoreASM Specification

Once the engine is initialized and loaded, it is ready to commence the execution of a simulated run. The engine waits for a *STEP* command, via the Control API, from the environment (e.g., an interactive GUI or a debugger), to start the actual computation of a step (Figures 4.6 to 4.9⁹); the receipt of this command results in a switch to mode *Starting Step*, and transfers control to the Scheduler.

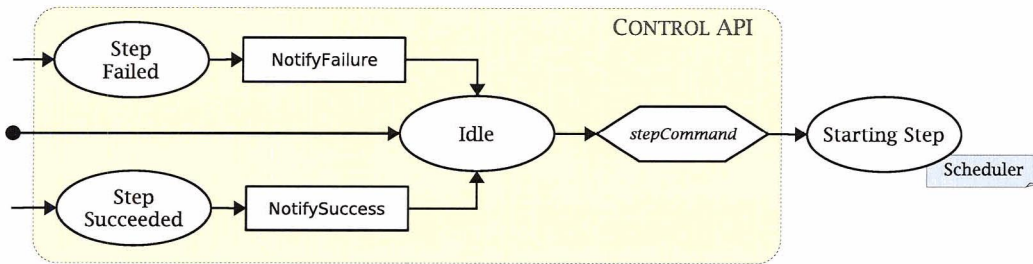


Figure 4.6: Control State ASM of a STEP command: Control API Module

The *StartStep* rule in the Scheduler initializes *updateSet* (the set of computed updates for the step), *agentSet* (the current set of active agents of the simulated machine), and *selectedAgentsSet* (the set of agents selected to perform computation in the current step). Then via the *RetrieveAgents*, the *selectedAgentsSet* is then assigned the value of *agents* from

⁹These figures were adapted from [23] by permission.

the current simulated state. To retrieve information regarding the value of a location, the Abstract Storage module must be queried; this interaction is modelled using an abstract function $getValue(l)$ which takes a location l and retrieves the value of the location from the simulated state. The notation "term" refers to a function or predicate named $term$ in the simulated machine. The mode is then changed to *Selecting Agents*.

Scheduler

StartStep \equiv

```

updateSet := { }
agentSet := undef
selectedAgentsSet := { }
    
```

RetrieveAgents \equiv

```

agentSet := getValue("agents", \)
    
```

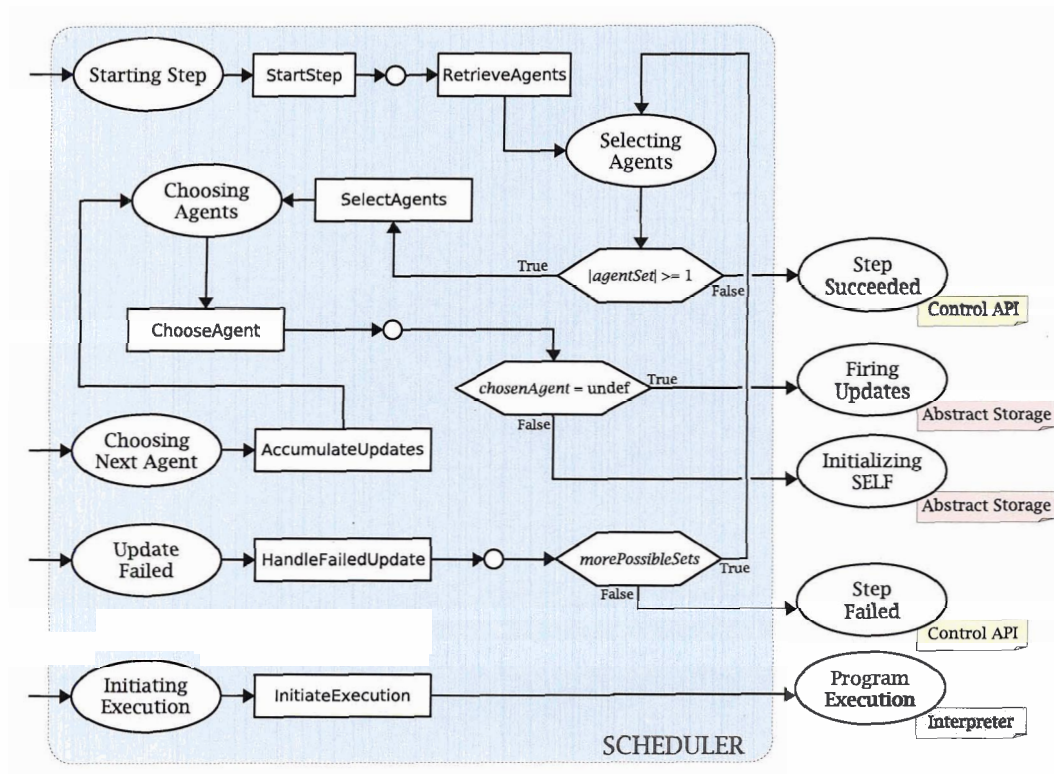


Figure 4.7: Control State ASM of a STEP command : Scheduler

When in the *Selecting Agents* mode, if there are no agents to execute, the step of the simulated machine is considered to be complete; otherwise, the **SelectAgents** rule selects a set of agents to perform computation in this step, and control moves to the *Choosing Agents* mode. Then the **ChooseAgent** rule selects an agent from this set and changes the mode to *Initializing SELF*, which leads to the execution of the **SetChosenAgent** and **GetChosenProgram** rules in the Abstract Storage module. Once the execution of the agent is concluded, the updates computed are collected by the **AccumulateUpdates** rule in the *Choosing Next Agent* mode. The engine returns to the *Choosing Agent* mode until all selected agents have been executed.

Scheduler

SelectAgents \equiv

```
choose  $s$  with  $s \subseteq agentSet \wedge |s| \geq 1$  do
  selectedAgentsSet :=  $s$ 
```

ChooseAgent \equiv

```
choose  $a$  in selectedAgentsSet do
  remove  $a$  from selectedAgentsSet
  chosenAgent :=  $a$ 
ifnone
  chosenAgent := undef
```

AccumulateUpdates \equiv

```
add updates(root(chosenProgram)) to updateSet
```

Before the execution of any agent, the engine enters the mode *Initializing SELF* in Abstract Storage. In this mode, the chosen agent is set (by assigning it to the distinguished variable *self* in the simulated state), and the program associated with the chosen agent is retrieved (by accessing *program(self)* in the simulated state). Control then returns to the Scheduler, going into mode *Initiating Execution*.

Abstract Storage

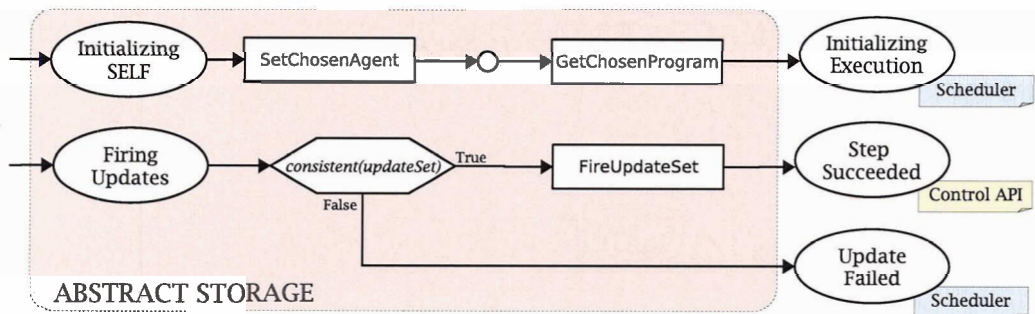
SetChosenAgent \equiv $\text{SetValue}(\langle \text{"self"}, \langle \rangle \rangle, \text{chosenAgent})$ **GetChosenProgram** \equiv $\text{chosenProgram} := \text{getValue}(\langle \text{"program"}, \langle \text{"self"} \rangle \rangle)$ 

Figure 4.8: Control State ASM of a STEP command : Abstract Storage

The execution of the chosen agent's program is first initialized in the *Initiating Execution* mode in the Scheduler, and then begins in the *Program Execution* mode in the Interpreter. Evaluation of each program results in updates being produced and collected in the *updateSet*. When all selected agents have completed their computation, control moves to the *Firing Updates* mode where application of updates is attempted. If the update set is consistent, it will be applied to the current state; if it is inconsistent, it will result in a failed update.

During interpretation of a program, values, updates and locations computed are associated with nodes of its AST. Just before commencing the interpretation of a program, the *InitiateExecution* rule removes all information resulting from the previous interpretation of the chosen program, and sets a pointer (always holding the current position in the tree — denoted by the nullary function *pos*) to the root node of the tree representing the program of the chosen agent.

Scheduler

InitiateExecution \equiv $\text{ClearTree}(\text{root}(\text{chosenProgram}))$ $\text{pos} := \text{root}(\text{chosenProgram})$

A full specification of the method used for interpretation of concrete rules and expressions is presented in greater detail in Section 4.2. However, the way in which the Interpreter interacts with plug-ins to delegate interpretation of the AST is discussed here. This offloading of interpretation is done in the *Program Execution* mode, where the `ExecuteTree` rule (found in Specification 4.1) is repeatedly executed.

It was mentioned in Section 4.1.1 that nodes of the parse tree corresponding to syntax provided by a plug-in are annotated with a plug-in identifier; a special oracle, the $plugin(node)$ function, is used to abstract from the details of how this annotation is implemented. When a node is found to refer to a particular plug-in, control over its interpretation is given to the plug-in. This is accomplished by first using the $pluginRule$ function to retrieve the underlying machine rule for the plug-in, and then executing this rule.

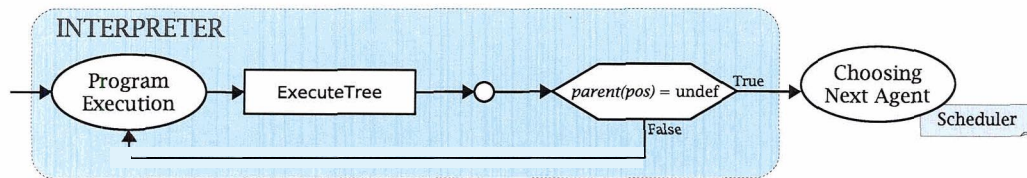


Figure 4.9: Control State ASM of a STEP command : Interpreter

However, nodes not associated with any plug-in are interpreted by the kernel via the `KernelInterpreter` (see Specification 4.2 in Section 4.2). Results of the interpretation of node pos are stored alongside the node, and are accessed by three functions in the underlying machine: $value(pos)$ holds the value computed for an expression node, $updates(pos)$ holds the set of updates generated by a rule node, and $loc(pos)$ holds the location denoted by the node (which is used as LHS-value within the **update**-rule). More precise definitions of these can be found in Section 4.2.1.

Interpreter
<pre> ExecuteTree \equiv if \negevaluated(pos) then if plugin(pos) \neq undef then let R = pluginRule(plugin(pos)) in R else KernelInterpreter else if parent(pos) \neq undef then pos := parent(pos) </pre>

Specification 4.1: The ExecuteTree rule in the Interpreter module.

When all selected agents have been executed, and the engine is in the *Choosing Agents* mode of the Scheduler, the update instructions produced by all agents will have been collected in *updateSet*. At this point, control will move to the *Firing Updates* mode in Abstract Storage. The consistency of the update set is queried using the *isConsistent* function provided by Abstract Storage (see Appendix A). Abstract Storage provides the *SetValue* rule, which applies a single update to the simulated state. If the update set is consistent, the next state is obtained by the *FireUpdateSet* rule, which uses this interface to Abstract Storage to apply all updates in the update set.

Abstract Storage

```

FireUpdateSet  $\equiv$ 
  forall (l, v)  $\in$  updateSet do
    SetValue(l, v)

```

However, if an inconsistent update is produced, the simulated machine provides an indication of failure by changing its mode to *Update Failed*. The *HandleFailedUpdate* rule in the Scheduler module then attempts to select a previously untested subset of agents for execution, and the step is re-initiated. The process is repeated until one of two conditions is met: either a consistent set of updates is produced, in which case the engine moves to the *Step Succeeded* mode of the Control API; all possible combinations of agents have

been exhausted, which alternately results in a switch to the *Step Failed*. In both cases, the environment is notified of the outcome via the `NotifySuccess` and `NotifyFailure` rules in the Control API. The engine then returns to the *Idle* mode awaiting further commands from the environment.

4.1.3 The Kernel and Plug-ins

A key feature of CoreASM is its extensibility; only the most fundamental DASM functionality is provided by the kernel (see Figure 4.2).

As the state of the simulated machine is defined by functions and universes, both domains of functions and universes are included in the kernel. Because universes are represented by their characteristic functions, the Boolean domain and its two elements are included in the kernel. As all basic functions are partial (resulting in *undef*, when a function location is not specified), the distinguished element *undef* is included in the kernel. Every rule of a CoreASM specification is represented as an element in the simulated machine, and as such the domain of rules is included in the kernel.

Only two rules are defined in the kernel: the update-rule and the import-rule. Without updates being generated, there would be no way of making a transition to a new state, making the **update**-Rule a necessity. The **import**-rule is important as it has privileged access to the reserve.

Finally function term evaluation and local variable evaluation, as well as named-rule execution is included in the kernel, as these are required to execute a step of any structured ASM specification.

All other functionality is provided by plug-ins. Many common rule forms coming from Basic ASMs and Turbo ASMs, common background classes such as numbers and sets, and operations involving elements of these sorts, are included in the standard library of plug-ins which is automatically loaded for use with all CoreASM specifications. The architecture supports the following kinds of extensions, of which a plug-in may provide any or all:

- The addition of background classes containing elements of a certain sort. This would require:
 - (i) An extension to the Parser defining the concrete syntax (literals, static functions, etc.) needed for working with elements of the background.

- (ii) An extension to Abstract Storage providing encoding and decoding functions for representing elements of the background for storage purposes.
- (iii) An extension to the Interpreter providing the semantics for all the literals defined in the background.
- The definition of additional operators. The plug-in must provide:
 - (i) An extension to the Parser defining the concrete syntax all of operators provided.
 - (ii) An extension to the Interpreter providing the semantics for all the operators defined.
- The definition of additional rule forms. This would require:
 - (i) An extension to the Parser defining the concrete syntax of the rule form.
 - (ii) An extension to the Interpreter defining the semantics of the rule form.

Extension plug-ins which are not distributed as part of the standard library, must be explicitly imported into an ASM specification by a **use** directive.

4.2 The CoreASM Language

In this section, CoreASM language syntax and semantics are modelled through specification of their interpretation. The notation used in the specification of the Interpreter will be first introduced. A number of kernel specific constructs are presented, followed by a selection of rules and operators present in the standard library.

4.2.1 Notation

The Interpreter is specified as a collection of rules which mimic evaluation through traversal of an AST, producing a combination of value, location, and updates resulting from the evaluation of nodes. The following assumptions are made:

1. Nodes in the tree are in the domain of the following functions:
 - $first : \text{NODE} \rightarrow \text{NODE}$, $next : \text{NODE} \rightarrow \text{NODE}$, $parent : \text{NODE} \rightarrow \text{NODE}$ are static functions that facilitate tree navigation; by using these functions, the Interpreter

can access all the children nodes of a given node, or go back to its parent, (see Figure 4.3 for reference).

- $class : \text{NODE} \rightarrow \text{CLASS}$ returns the syntactical class of a node (i.e., used to classify and identify the node when its token is not sufficient).
- $token : \text{NODE} \rightarrow \text{TOKEN}$ returns the syntactical token which the node represents (e.g., either a keyword, an identifier, or a literal value).
- $\llbracket \cdot \rrbracket : \text{NODE} \rightarrow \text{LOC} \times \text{UPDATES} \times \text{ELEMENT}$ holds the result of the interpretation a node, given by a triple formed by a location (i.e. the LHS-value of an expression, when it is defined), a multiset of update instructions, and a value (i.e. the RHS-value of an expression)¹⁰. Properties of these triples may be established through the following derived functions:

- $loc : \text{NODE} \rightarrow \text{LOC}$ returns the location (LHS-value) associated with the given node, i.e. $loc(n) \equiv \llbracket n \rrbracket \downarrow 1$.
- $updates : \text{NODE} \rightarrow \text{UPDATES}$ returns the updates associated to the given node, i.e. $updates(n) \equiv \llbracket n \rrbracket \downarrow 2$.
- $value : \text{NODE} \rightarrow \text{ELEMENT}$ returns the value (RHS-value) associated with the given node, i.e. $value(n) \equiv \llbracket n \rrbracket \downarrow 3$.
- $evaluated : \text{NODE} \rightarrow \text{BOOLEAN}$ indicates if a node has been fully evaluated, where

$$evaluated(n) \equiv \llbracket n \rrbracket \neq undef$$

- $plugin : \text{NODE} \rightarrow \text{PLUGIN}$ is the plug-in associated with the node, and hence responsible for parsing and evaluating it.

2. At all times the Interpreter's current position in the tree is kept in a distinguished variable pos .

3. A form of pattern matching which allows for concisely specifying complex conditions on the nodes. In particular:

- arbitrary nodes are denoted with $\boxed{?}$.

¹⁰Organization of the triple is intended to be mnemonic with respect to an **update**-rule, the LHS-value being in the leftmost position, and the RHS-value in the rightmost position of a triple. Updates which would be produced by the evaluation of a rule, reside in the central position.

- arbitrary unevaluated nodes are denoted with \square ; as an aid to the reader, the semantically equivalent \square_e , \square_r , and \square_l denote unevaluated nodes whose evaluation is expected to result, respectively, in a value (from an expression), a set of updates (from a rule), and a location.
- an identifier node is denoted with x .
- an evaluated expression node (that is, a node whose *value* — resultant value — is not *undef*) is denoted with v ; an evaluated statement node (a node whose *updates* — resultant set of updates — are not *undef*) is denoted with u ; an evaluated expression for which a location has been computed (a node whose *loc* — resultant location — is not *undef*) is denoted with l . Subscripts may be appended to these variables, or different names for the variables may be used, for special cases and are discussed as appropriate.
- prefixed Greek letters are used to denote positions in the AST (typically children of the current node as denoted by *pos*) as in **if** ${}^\alpha e$ **then** ${}^\beta r$ where α and β denote, respectively, the condition node and the then-part node of an **ifelse**-rule being interpreted.
- *pattern-action rules* (PA rules) of the form

$$(\textit{pattern}) \rightarrow \textit{actions}$$

are to be read as

if *conditions* **then** *actions*

where the meanings of *conditions* and *actions* are derived using the notation convention informally described above, and formally specified in Table 4.1. In the action part of such a PA rule: any unquoted and unbound occurrence of l is to be interpreted by the reader as the *loc* of the corresponding node; any unquoted and unbound occurrence of v is to be interpreted by the reader as the *value* of the corresponding node; any unquoted and unbound occurrence of u as the *updates* of the corresponding node; and any unquoted and unbound occurrence of x as the *token* of the corresponding node.

In summary, the pattern part of a PA rule depicts, using the given notation, conditions on the current node and its subtree, including the status of node and subtree

Abbreviation in PA-rule	Meaning in Pattern part	Meaning in Action part
α, β etc.	$first(pos), next(first(pos)),$ etc.	$first(pos), next(first(pos)),$ etc.
$\alpha \boxed{?}$	$class(\alpha) \neq ld$	
$\alpha \boxed{}$	$class(\alpha) \neq ld \wedge \neg evaluated(\alpha)$	
$\alpha \boxed{e}, \alpha \boxed{r}, \alpha \boxed{l} *$	$class(\alpha) \neq ld \wedge \neg evaluated(\alpha)$	
αx	$class(\alpha) = ld$	$token(\alpha)$
αv	$value(\alpha) \neq undef$	$value(\alpha)$
αu	$updates(\alpha) \neq undef$	$updates(\alpha)$
αl	$loc(\alpha) \neq undef$	$loc(\alpha)$

* These symbols are semantically equivalent to the $\boxed{}$ symbol, however as a visual cue to the reader, the embedded letters express the intended result of their evaluation.

Table 4.1: Abbreviations and their meanings in syntactic PA rules.

evaluation. The action part describes the actions to be taken by the underlying machine if the condition expressed in the pattern part is met. Table 4.2 illustrates how this notation can be translated into ASM rules for the interpretation of an **ifelse**-rule. Multiple PA rules may be used to define the semantics of a single syntactic form in the simulated machine; each PA rule of a syntactic form's definition represents a different stage in the interpretation of that form's node (i.e. before/after children of the node are interpreted).

4. The value of local variables (e.g., those defined in **let**-rules) is maintained by a global dynamic function of the form $env : \text{TOKEN} \rightarrow \text{ELEMENT}$.
5. The static function $bkg : \text{ELEMENT} \rightarrow \text{BACKGROUND}$ provides, for any arbitrary value v , the background class of the value or $undef$ if the value is not of any specific sort (e.g. if the **ELEMENT** was imported from the reserve).

It is important to notice that nodes are interpreted only when $evaluated(pos)$ is not *true* (i.e. if the current node has not been evaluated — see rule **ExecuteTree** described in Specification 4.1). Control moves from node to node either by:

- Explicitly setting the value of pos to the node to which control should be given.
- Setting $\llbracket pos \rrbracket$ to a value which is not $undef$; control then is given to the parent of pos by the **ExecuteTree** rule.

PA-rule notation	Corresponding ASM rule
$(\text{if } \alpha \boxed{e} \text{ then } \beta \boxed{r}) \rightarrow pos := \alpha$	if $class(pos) \neq \text{ld}$ $\wedge token(pos) = \text{IfThen}$ $\wedge class(first(pos)) \neq \text{ld}$ $\wedge \neg evaluated(first(pos))$ $\wedge class(next(first(pos))) \neq \text{ld}$ $\wedge \neg evaluated(next(first(pos)))$ then $pos := first(pos)$
$(\text{if } \alpha v \text{ then } \beta \boxed{r}) \rightarrow \text{if } v = \text{tt} \text{ then } \dots$	if $class(pos) \neq \text{ld}$ $\wedge token(pos) = \text{IfThen}$ $\wedge value(first(pos)) \neq \text{undef}$ $\wedge class(next(first(pos))) \neq \text{ld}$ $\wedge \neg evaluated(next(first(pos)))$ then if $value(first(pos)) = \text{tt}$ then \dots
$(\text{if } \alpha v \text{ then } \beta u) \rightarrow \dots$	if $class(pos) \neq \text{ld}$ $\wedge token(pos) = \text{IfThen}$ $\wedge value(first(pos)) \neq \text{undef}$ $\wedge updates(next(first(pos))) \neq \text{undef}$ then \dots

Table 4.2: Examples of how PA rule notation is translated into ASM rules.

Thus, the interpretation of any given node should result in all needed subtrees being first evaluated (by moving control to children nodes via explicit assignment to pos), followed by the evaluation of the node itself using information from evaluated children (the result of the evaluation stored in $\llbracket pos \rrbracket$), causing control to be return back to its parent. Using notation in Table 4.2, it is easy to visualize this process by the progressive substitution of evaluated u nodes for unevaluated \boxed{r} nodes, and of v or l nodes for unevaluated \boxed{e} nodes.

4.2.2 Kernel Interpreter

The portions of the CoreASM language provided by the kernel are interpreted via the `KernelInterpreter` rule shown in Specification 4.2 (via `ExecuteTree` found in Specification 4.1). This rule causes the parallel execution (using a **block-rule**) of PA rules, resulting in the correct interpretation of a kernel node based on pattern conditions.

Interpreter
<p>KernelInterpreter \equiv</p> <p>$(\text{pattern}_1) \rightarrow \text{actions}_1$</p> <p style="text-align: center;">⋮</p> <p>$(\text{pattern}_n) \rightarrow \text{actions}_n$</p>

Specification 4.2: The `KernelInterpreter` rule in the Interpreter module.

As mentioned earlier, the kernel provides the Boolean universe and its literals, the *undef* element, function evaluation, **named-rule** call, the **update-rule**, and the **import-rule**. Here we introduce a subset of this functionality, namely the evaluation of distinguished element literals, and the **update** and **import** rules.

The evaluation of the literals simply results in the retrieval of the appropriate element from the state of the simulated machine and setting it as the value of the node:

	Kernel Interpreter: Literals
$(\text{true}) \rightarrow \llbracket pos \rrbracket := (\text{undef}, \text{undef}, \text{tt})$	
$(\text{false}) \rightarrow \llbracket pos \rrbracket := (\text{undef}, \text{undef}, \text{ff})$	
$(\text{undef}) \rightarrow \llbracket pos \rrbracket := (\text{undef}, \text{undef}, \text{uu})$	

Note that *tt*, *ff*, and *uu* represent the distinguished elements *true*, *false*, and *undef* in the simulated machine, respectively.

An **update**-rule (see the “:=” node in Figure 4.3) is interpreted as follows:

Kernel Interpreter: Update Rule

$$\begin{aligned}
 \llbracket \alpha \square := \beta \square \rrbracket &\rightarrow \text{choose } \tau \in \{\alpha, \beta\} \text{ with } \neg \text{evaluated}(\tau) \\
 &\quad pos := \tau \\
 &\text{ifnone} \\
 &\quad \text{if } loc(\alpha) \neq undef \\
 &\quad \quad \llbracket pos \rrbracket := (undef, \{(loc(\alpha), value(\beta))\}, undef) \\
 &\quad \text{else} \\
 &\quad \quad \text{Error('Cannot update a non-location.')}
 \end{aligned}$$

Notice that only child nodes evaluating to locations may be assigned values. The evaluation of subtrees representing functions (see the “*func()*” node in Figure 4.3) should result in both a location and value, the location being used if the function term is on the left-hand side of an **update**-rule, and the value being used if on the right-hand side.

The **import**-rule is interpreted as follows:

Kernel Interpreter: Import Rule

$$\begin{aligned}
 \llbracket \text{import } \alpha x \text{ do } \beta \square \rrbracket &\rightarrow \text{let } e = \text{new}(\text{ELEMENT}) \text{ in} \\
 &\quad env(x) := e \\
 &\quad pos := \beta \\
 \\
 \llbracket \text{import } \alpha x \text{ do } \beta u \rrbracket &\rightarrow env(x) := undef \quad // \text{No nesting} \\
 &\quad \llbracket pos \rrbracket := (undef, u, undef)
 \end{aligned}$$

Here a new ELEMENT (being an element of the simulated machine universe) is created and assigned to the value of the identifier *x* in the local environment. The rule part is then evaluated in this new environment by setting *pos* to β . Once the evaluation of the rule part subtree is complete, the local value of the given identifier is erased. This illustrates how the local environment of the simulated machine is modified using the *env* function of the underlying machine, thus only allowing the use of a variable in the proper scope.

4.2.3 Extension Interpretation

When the kernel is not responsible for the interpretation of the node, the interpretation rule from the plug-in (from the standard library or otherwise) associated with the node is

executed. In the `ExecuteTree` in Specification 4.1, the `pluginRule` function returns a rule of the form

ArbitraryPluginInterpreter \equiv

$\langle \langle pattern_1 \rangle \rangle \rightarrow actions_1$

\vdots

$\langle \langle pattern_n \rangle \rangle \rightarrow actions_n$

Arbitrary Plugin Interpreter

which is provided by the plugin, and is similar in structure to the `KernelInterpreter` rule introduced in Specification 4.2. Here we present some rules and operators which are provided by plug-ins of the CoreASM standard library. In particular, we will provide PA rules for the **par**, **ifelse**, and **seq** rules, followed by the equals (=), Boolean *AND* (\wedge), and numerical unary minus ($-$) operators.

Rule Extensions

The **par**-rule is interpreted as follows¹¹:

$\langle \langle \mathbf{par} \ \lambda_1 \langle ? \rangle \ \dots \ \lambda_n \langle ? \rangle \rangle \rangle \rightarrow \mathbf{choose} \ i \in [1..n] \ \mathbf{with} \ \neg \mathit{evaluated}(\lambda_i)$
 $\quad \quad \quad \mathit{pos} := \lambda_i$
 $\quad \quad \quad \mathbf{ifnone}$
 $\quad \quad \quad \llbracket \mathit{pos} \rrbracket := (\mathit{undef}, \bigcup_{i \in [1..n]} \mathit{updates}(\lambda_i), \mathit{undef})$

Par Rule

Notice that rules in a block are executed in an order which is unspecified, the final result being the union of all the updates produced by evaluation of these rules.

In the **ifelse**-rule interpretation, any guard resulting in a **tt** results in execution of the **if** part, otherwise the **else** part being executed:

¹¹We provide interpretation for an n -rule block where $n \geq 1$. Also note that here we are disregarding the scope constructors provided by the grammar — indentation or a matching **endpar** are often used.

	IfElse Rule
$\langle \text{if } \alpha \underline{e} \text{ then } \beta \underline{r} \rangle \rightarrow pos := \alpha$	
$\langle \text{if } \alpha v \text{ then } \beta \underline{r} \rangle \rightarrow \text{if } v = \text{tt} \text{ then } pos := \beta \text{ else } \llbracket pos \rrbracket := (\text{undef}, \{\}, \text{undef})$	
$\langle \text{if } \alpha v \text{ then } \beta u \rangle \rightarrow \llbracket pos \rrbracket := (\text{undef}, u, \text{undef})$	
<hr/>	
$\langle \text{if } \alpha \underline{e} \text{ then } \beta \underline{r} \text{ else } \gamma \underline{r} \rangle \rightarrow pos := \alpha$	
$\langle \text{if } \alpha v \text{ then } \beta \underline{r} \text{ else } \gamma \underline{r} \rangle \rightarrow \text{if } v = \text{tt} \text{ then } pos := \beta \text{ else } pos := \gamma$	
$\langle \text{if } \alpha v \text{ then } \beta u \text{ else } \gamma \underline{r} \rangle \rightarrow \llbracket pos \rrbracket := (\text{undef}, u, \text{undef})$	
$\langle \text{if } \alpha v \text{ then } \beta \underline{r} \text{ else } \gamma u \rangle \rightarrow \llbracket pos \rrbracket := (\text{undef}, u, \text{undef})$	

Recall that the **seq**-rule models sequential execution of rules. The semantics of sequential execution requires us to model the effect of evaluating the second rule in the state produced by applying the updates produced by the first rule. However this application of updates must be simulated (i.e. not really modifying the current state). Using a stack of states, the manipulation of a temporary copy of the state is achieved. That stack is managed through three rules: **PushState** puts a copy of the current state on top of the stack, **PopState** retrieves the state from the top of the stack (thus discarding the temporary state), and **Apply**(u) applies the updates in the update set u to the state residing on top of the stack. Formal definitions for these macros are given in Appendix A. Using this functionality the **seq**-rule can be specified as shown in Specification 4.3 below

	Seq Rule
$\langle \alpha \underline{r}_1 \text{ seq } \beta \underline{r}_2 \rangle \rightarrow pos := \alpha$	
$\langle \alpha u_1 \text{ seq } \beta \underline{r}_2 \rangle \rightarrow \text{if } \text{isConsistent}(u_1) \text{ then}$	
PushState	
Apply (u_1)	
$pos := \beta$	
else	
$\llbracket pos \rrbracket := (\text{undef}, u_1, \text{undef})$	
$\langle \alpha u_1 \text{ seq } \beta u_2 \rangle \rightarrow \text{PopState}$	
$\llbracket pos \rrbracket := (\text{undef}, u_1 \oplus u_2, \text{undef})$	

Specification 4.3: The PA rule for interpretation of the **seq**-rule.

where the \oplus operator is as defined in Section 2.2.

Operator Extensions

The CoreASM equivalence operator shown in Specification 4.4 uses the CoreASM wide notion of element equality (see the definition of *equal* in Appendix Section A.1.1). Two elements are considered to be equal iff at least one of their backgrounds regards the values as equal. As such, each background implementation must provide an equality function for elements originating from them.

Equivalence Operator
$\llbracket \alpha \text{?} = \beta \text{?} \rrbracket \rightarrow \text{choose } \lambda \in \{\alpha, \beta\} \text{ with } \neg \text{evaluated}(\lambda)$ $\quad \text{pos} := \lambda$ ifnone $\quad \text{let } e_1 = \text{value}(\alpha), e_2 = \text{value}(\beta) \text{ in}$ $\quad \text{if } \text{equal}(e_1, e_2) \text{ then}$ $\quad \quad \llbracket \text{pos} \rrbracket := (\text{undef}, \text{undef}, \text{tt})$ $\quad \text{else}$ $\quad \quad \llbracket \text{pos} \rrbracket := (\text{undef}, \text{undef}, \text{ff})$

Specification 4.4: PA rule depicting semantics of the equivalence operator.

The Boolean AND operator results in *tt* if both of its operands evaluate to *tt*, and *ff* otherwise:

Boolean Operator: AND
$\llbracket \alpha \text{?} \wedge \beta \text{?} \rrbracket \rightarrow \text{choose } \lambda \in \{\alpha, \beta\} \text{ with } \neg \text{evaluated}(\lambda)$ $\quad \text{pos} := \lambda$ ifnone $\quad \text{if } (\text{value}(\alpha) = \text{tt}) \wedge (\text{value}(\beta) = \text{tt}) \text{ then}$ $\quad \quad \llbracket \text{pos} \rrbracket := (\text{undef}, \text{undef}, \text{tt})$ $\quad \text{else}$ $\quad \quad \llbracket \text{pos} \rrbracket := (\text{undef}, \text{undef}, \text{ff})$

The unary numerical negation operator would be interpreted as follows:

Numerical Operator: Unary Minus

$\llbracket -^\alpha \square \rrbracket \rightarrow pos := \alpha$

$\llbracket -^\alpha v \rrbracket \rightarrow \mathbf{let } r = negNum(v) \mathbf{ in}$
 $\quad \llbracket pos \rrbracket := (undef, undef, r)$

Here the *negNum* function results in an element representing the numerical negation of the operand.

“The world is moved along, not only by the mighty shoves of its heroes, but also by the aggregate of the tiny pushes of each honest worker.”

— Helen Keller

Chapter 5

Distributed Incremental Change with Aggregation

In Section 2.7 we introduced the notion of incremental change to elements that have an internal structure. In particular, we briefly described the general mathematical framework of Gurevich’s *partial updates* and *integration* for distributed incremental change of elements. In this chapter we formally specify how we incorporate this functionality into the CoreASM engine, while ensuring future extensibility.

While our pragmatic approach is similar in spirit to Gurevich’s mathematical framework, it is not the same. Our method of representing and resolving simultaneous incremental change is different, and thus, the terminology we use different as well. Where Gurevich refers to partial updates as updates representing a partial modification of an element, we use the term *incremental updates*. Where Gurevich refers to the process of combining partial updates into a total update as integration, the process which we use in CoreASM is called *aggregation*.

In this chapter we first describe how incremental updates are represented in the simulated machine, and how aggregation is incorporated into a step of the simulated machine. This is followed by a description of how Turbo ASM sequential composition is accomplished with our approach. The chapter ends with a formal specification of the entire *Set Plug-in* which includes set-related background, rule, operator and aggregation extensions to the CoreASM engine.

5.1 Aggregation and Incremental Updates

In this section, we introduce incremental updates and aggregation into the CoreASM engine architecture described in Chapter 4. In particular we discuss how the embodiment of aggregation affects the engine architecture and a CoreASM step as a whole.

5.1.1 Rules and Their Side Effects

As each rule of a CoreASM specification is executed by the Interpreter, it is expected to produce a (potentially empty) set of updates, each update being viewed as a 2-tuple expected to consist of a location and a value:

$$\langle \text{LOC}, \text{ELEMENT} \rangle$$

The union of all of these sets returned by rules during a single step of the simulated machine constitute the update set for a CoreASM step.

However, the possibility of having elements within the simulated machine which are themselves based on axioms and structure (e.g. sets, maps, trees) and whose internal structures may also be updated by rules, requires the CoreASM have facilities to handle such incremental updates. Recall that Gurevich's partial updates consist of a location and a particle. The particle represents the partial modification to be made to the element at the given location; the particle is a mathematical function in which the entire incremental change is encoded. Notice that the essential function of a particle in a partial update is to represent the type of incremental change to perform, and a value associated to that change; for instance the `setaddmsgx` particle, introduced in our message passing protocol example in Section 2.7, represents the addition of an element to a set and the value of the element to be added which is `msgx`.

To accommodate the representation of incremental change into CoreASM, we allow rules to return *update instructions*, rather than updates; like updates, they consist of location and value, but they also include an *action* to be performed on the element at the the given location. Update instructions are viewed as a 3-tuple of the form:

$$\langle \text{LOC}, \text{ELEMENT}, \text{ACTION} \rangle$$

The combination of the value and action represent the intended incremental modification to be made to the element residing at the given location. Update instructions containing

incremental modification actions are referred to as *incremental updates*. Going back to our example, the incremental update resulting in the addition of element msg_x to a set at location l would produce an update instruction of the form:

$$\langle l, msg_x, setAddAction \rangle$$

where $setAddAction \in \text{ACTION}$ and ACTION is the domain of all actions supported by the simulated machine.

Regular Updates

For the sake of homogeneity we require the **update**-rule to return update instructions as well. However the action for such *regular updates* is always $updateAction \in \text{ACTION}$.¹

$$\langle \text{LOC}, \text{ELEMENT}, updateAction \rangle$$

Update and Update Instruction

When discussing ASMs, the term *update* is typically used to refer to both the act of modifying a location, as well as the data structure representing an update. With the introduction of the *update instruction*, when discussing the CoreASM machine we use the term update to refer to the act of modifying a location, whereas the term update instruction is used to refer to the data structure representing an update of any kind (i.e. regular or incremental update); however, at times we also use these terms interchangeably when the difference between them is irrelevant.

5.1.2 Update Instruction Notation

All update instructions have the following functions defined over them:

- $\langle \cdot \rangle : \text{UPDATEINST} \rightarrow \text{LOC} \times \text{ELEMENT} \times \text{ACTION}$ holds the constituents of the update instruction given by a triple formed by a location, a value, and an action to be performed. We access elements and establish properties of such triples through the following derived functions:

¹This follows Gurevich's approach, where a total update to a location results in a partial update containing an *overwrite* particle. See Section 2.7 for more information.

- $uiLoc : \text{UPDATEINST} \rightarrow \text{LOC}$ returns the location associated with the given update instruction, i.e. $uiLoc(ui) \equiv \langle\langle ui \rangle\rangle \downarrow 1$.
- $uiVal : \text{UPDATEINST} \rightarrow \text{ELEMENT}$ returns the value associated with the given update instruction, i.e. $uiVal(ui) \equiv \langle\langle ui \rangle\rangle \downarrow 2$.
- $uiAction : \text{UPDATEINST} \rightarrow \text{ACTION}$ returns the action associated with the given update instruction, i.e. $uiAction(ui) \equiv \langle\langle ui \rangle\rangle \downarrow 3$.
- $aggStatus : \text{UPDATEINST} \times \text{PLUGIN} \rightarrow \text{FLAG}$ indicates the aggregation status of an update instruction, as set by a given aggregator plug-in; $\text{FLAG} = \{successful, failed\}$. If an update instruction has not been processed by a plug-in, *undef* is returned. Note that the purpose this function will become more clear in subsequent sections.

5.1.3 A CoreASM Step

The computation of a single step of an ASM program can be summarized very simply as follows:

1. Execute the program rule and collect the updates into a set.
2. If update set is consistent, apply the updates.

Notice that step 1 of this process is completed in the Scheduler, upon transition to the *Firing Updates* mode in the Interpreter (see Figures 4.7 and 4.8). Step 2 is accomplished in the *Firing Updates* mode of the engine, where the consistency of the update set is queried with the *isConsistent* function, and if *updateSet* is consistent, all updates are fired by the *FireUpdateSet* rule.

However, with the introduction of incremental updates, the process of creating the final update set requires additional work. Update instructions are collected into an *update multiset*² stored in the function *updateInstructions*. Once the execution of the program rule is complete, all update instructions pertaining to a particular location are *aggregated* into one single update per location. *Aggregation* is the process of combining all update instructions affecting a single location of a machine into one single update called the *resultant update*. The *aggregation phase* of a CoreASM step performs aggregation on all locations affected by

²Recall from Section 2.7, that Gurevich also uses multisets to hold partial updates, as multiplicity may be important in their integration.

the step. Note that resultant updates cannot and should not depend on the order in which all update instructions for a location are combined, as all updates producing them occur simultaneously according to ASM semantics; we shall explain this further in Section 5.1.6.

It is important to highlight the difference between regular, resultant, and basic updates. A *regular* update is a typical ASM update produced by an **update**-rule, whereas a *resultant* update is an ASM update produced by aggregating all incremental update instructions and all regular updates into one unified change affecting a single location of the machine. The word “resultant”, “regular” will be dropped when its’ meaning is obvious from context. An update is *basic* if every update operating on its location is regular, thus implying no aggregation need be performed on its location.

The aggregation phase results in an update set, consisting of basic updates and resultant updates. The traditional ASM step augmented with the aggregation phase is summarized as follows:

1. Execute the program rule and collect the update instructions into a multiset.
2. Aggregate the update instructions in the multiset, producing the update set.
3. If the aggregation phase is successful and update set is consistent, apply the updates.

In Figures³ 5.1 and 5.2 respectively, we show revised control state ASMs for the Scheduler and Abstract Storage modules depicting how the aggregation phase augments a CoreASM step.

When control is in the Scheduler (in the *Choosing Agents* mode) and all agents selected to execute in a step have been executed, control now moves to the *Aggregation* mode in Abstract Storage. Here the rule *AggregateUpdates* (which is formally defined in the next section) performs the aggregation of the multiset *updateInstructions*. When aggregation is complete, control moves to the *Firing Updates* mode where both update set consistency and aggregation consistency are confirmed before application of the update set.

³These figures were adapted from [23] by permission.

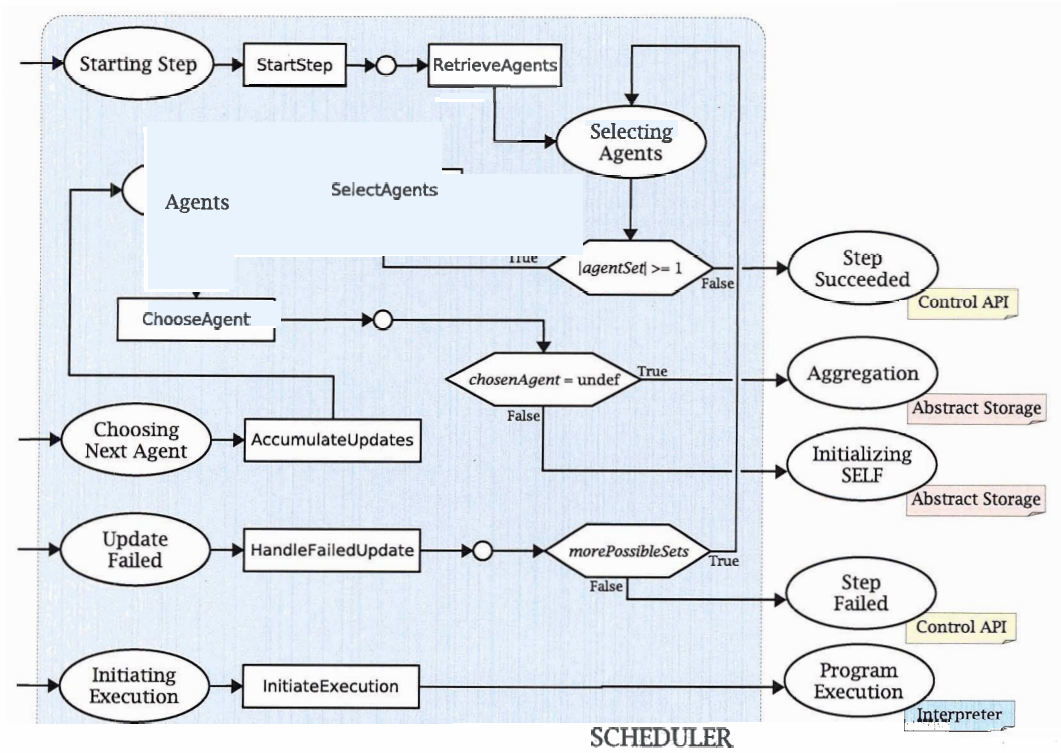


Figure 5.1: Revised control state ASM of a Step command with Aggregation : Scheduler

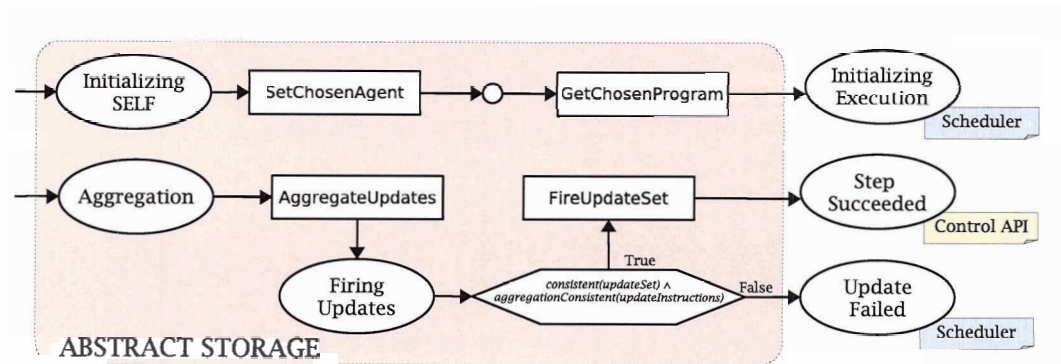


Figure 5.2: Revised control state ASM of a Step command with Aggregation : Abstract Storage

5.1.4 Responsibility for Aggregation

Background plug-ins, which extend CoreASM with a background class should provide all that is necessary to manipulate elements which originate from their background⁴. For backgrounds that consist of elements with internal structure that can be manipulated, background plug-ins provide rule forms that result in incremental update instructions, as well as provide an algorithm for aggregation. We call these plug-ins *aggregators* or *aggregator plug-ins*.

We say that an aggregator plug-in is responsible for:

- An action other than the *updateAction* action (see Section 5.1.5), if it is equipped to handle its aggregation.
- Aggregation of a given update instruction if the update instruction:
 - Contains an action for which the plug-in is responsible.
 - Contains an *updateAction* (making it a regular update) and there is another update instruction which it is responsible for that also operates on the the same location.
- A location if update instructions operating on that location are its responsibility.

Upon being called for aggregation, a plug-in will aggregate all update instructions for which it is responsible, flagging those update instructions it has processed. It is important to note that the order in which plug-ins are called to perform aggregation does not affect the resultant updates produced. Also note that the failure in aggregation of a single plug-in will not foil the aggregation attempts of other plug-ins. Upon completion of the aggregation phase, an update set is created from the union of resultant updates.

⁴While we expect a plug-in providing a background class to provide all that is necessary to manipulate elements of its background, there may be cases where it is more appropriate for functionality to be present in different plug-ins. Thus, we do not enforce this expectation.

AggregateUpdates

```
updateSet ← Aggregate(updateInstructions)
```

```
Aggregate(uMset : UPDATEMULTISET) ≡
```

```
let ap = {a | a ∈ PLUGIN ∧ aggregator(a)} in
```

```
forall p ∈ ap do
```

```
    resultantUpdates(p, uMset) ← InvokeAggregation(p, uMset)
```

```
seq
```

```
// Results in an update set
```

```
result :=  $\bigcup_{p \in ap}$  resultantUpdates(p, uMset)
```

```
InvokeAggregation(p : PLUGIN, uMset : UPDATEMULTISET) ≡
```

```
let R = aggregatorRule(p) in
```

```
result ← R(uMset)
```

The *resultantUpdates* function is used to collect resultant updates from plug-ins for a given multiset of update instructions, and the *aggregatorRule* function is expected to return the rule implementing the aggregation algorithm of the given plug-in. Note that the parameterized rule for aggregation, *Aggregate*, may be called on any update multiset; thus, it can be used by Turbo ASM rule-form implementations to perform aggregation on update multisets of their simulated steps. Also note that in *InvokeAggregation*, a plug-in aggregator rule is expected to accept a multiset as an argument, and its invocation should cause the return of its resultant updates.

5.1.5 Basic Update Aggregator

The keen observer will have noticed that once all aggregator plug-ins have completed aggregation successfully, the resultant update set will not contain basic updates (i.e. regular updates for locations which do not require aggregation). The *Basic Update Aggregator* solves this problem by masquerading as an aggregator plug-in and returning a set of all regular updates for locations which do not require any aggregation. It is defined as follows

```

BasicUpdateAggregator(uMset : UPDATEMULTISET)  $\equiv$ 
  result := { }
  seq
  forall ui  $\in$  uMset with uiAction(ui) = updateAction do
    if  $\nexists$  ui'  $\in$  uMset, uiLoc(ui) = uiLoc(ui')  $\wedge$  uiAction(ui')  $\neq$  updateAction then
      add ui to result
      aggStatus(ui, buPlugin) := successful

```

where *buPlugin* represents the Basic Update Aggregator as a plug-in, *buPlugin* \in PLUGIN and *aggregator*(*buPlugin*) = *true*. Evaluation of *aggregatorRule*(*buPlugin*) results in the rule BasicUpdateAggregator. The Basic Update Aggregator is called by Aggregate along with all aggregator plug-ins. Note that the Basic Update Aggregator flags all update instructions it processes with *successful*.

5.1.6 Plug-in Aggregation Consistency

While a plug-in is performing its aggregation on the multiset, it may encounter a situation where the update instructions for a given location that it is responsible for cannot be aggregated into a regular update. Such a situation occurs when one of the following holds:

- There are update instructions which make no semantic sense in context⁵. (e.g. the addition of an element to a set, on a location which contains no set element in the current state).
- The result of aggregation of a location depends on the order in which incremental update instructions for that location are combined. Recall that since incremental updates resulting from a single step of the machine occur the same time according to ASM semantics, the result of their aggregation must not be ambiguous for their aggregation to be consistent.

When the aggregation of all update instructions affecting a given location is deemed inconsistent, the following rule is called by the plug-in to flag all updates to the location as *failed*:

⁵Acceptable semantics of incremental updates, and the aggregation resulting from their update instructions, are defined by the aggregation algorithm which processes them.

```

HandleInconsistentAggregation( $l : \text{LOC}, uMset : \text{UPDATEMULTISET}, p : \text{PLUGIN}$ )  $\equiv$ 
  forall  $ui \in uMset$  with  $uiLoc(ui) = l$  do
     $aggStatus(ui, p) := failed$ 

```

Although aggregation for a single location may have failed, the aggregation of the rest of the update instructions the plug-in is responsible for would continue.

5.1.7 Aggregation Algorithms Provided

There are very few hard-and-fast requirements on the algorithm provided by an aggregator plug-in. It is expected to:

- Aggregate all update instructions in the update multiset that it is responsible for, and return the set of all its resultant updates.
- Determine if aggregation on a given location will result in inconsistency, and handle such inconsistencies appropriately.
- Flag all update instructions considered during its aggregation as either *successful* or *failed*.

The process of aggregation and consistency determination depends largely on the semantics of incremental updates for a given background and its elements. Axioms of the internal structure of the elements guide the plug-in writer in determining what is considered consistent (i.e. what makes sense), and what is not. In some cases, the multiplicity of an update instruction performed on a given location is important in determining the semantics of the incremental update: [38] gives the example of the background class of counters to illustrate this point. For this reason, the data structure used for collecting update instructions is a *multiset* rather than a set.

The freedom given to plug-ins in determining their own aggregation promotes the extensibility of the engine with background classes for the widest possible variety of sorts.

5.1.8 Aggregation Phase Consistency

Once the aggregation phase is complete, aggregation consistency can be checked with the following function:

- **derived** *aggregationConsistent* : UPDATEMULTISET \rightarrow BOOLEAN
returns *true* if aggregation was completed with consistency; *false* is returned otherwise.
It is defined as:

$$\begin{aligned} \text{aggregationConsistent}(uMset) \equiv \\ \text{allUpdatesProcessed}(uMset) \wedge \text{noAggregationFailures}(uMset) \end{aligned}$$

There are two conditions which must be met in order to ensure the consistency of aggregation:

1. All updates in the multiset should have been processed (and should have some status flag). Every update instruction should have either been processed by the Basic Update Aggregator, or an aggregator plug-in.

- **derived** *allUpdatesProcessed* : UPDATEMULTISET \rightarrow BOOLEAN
returns *true* if all update instructions have been processed; *false* is returned otherwise.
It is defined as:

$$\begin{aligned} \text{allUpdatesProcessed}(uMset) \equiv \\ \forall ui \in uMset, \exists p \in \text{PLUGIN}, \text{aggregator}(p) \wedge \text{aggStatus}(ui, p) \neq \text{undef} \end{aligned}$$

2. There should be no update instructions in the multiset which have been flagged as *failed*.

- **derived** *noAggregationFailures* : UPDATEMULTISET \rightarrow BOOLEAN
returns *true* if all locations were aggregated consistently; *false* is returned otherwise.
It is defined as:

$$\begin{aligned} \text{noAggregationFailures}(uMset) \equiv \\ \forall ui \in uMset, \nexists p \in \text{PLUGIN}, \text{aggregator}(p) \wedge \text{aggStatus}(ui, p) = \text{failed} \end{aligned}$$

When the aggregation phase is considered to be inconsistent, this constitutes a failed step of the simulated machine (as does an inconsistent update set).

Notice that aggregation is not considered to be inconsistent if update instructions have been successfully processed more than once, potentially by multiple plug-ins. In such a situation, each plug-in processing instructions for a location will produce a resultant update

for that location. This will not pose a problem if the two resultant updates do not conflict. However, if they do indeed conflict, this problem will be caught during the update set consistency check. Thus, multiple successfully processed update instructions are not always problematic and so a check for this situation is not incorporated into *aggregationConsistent* with the understanding that, if there is a problem, it will be caught during the update set consistency check.

5.2 Turbo ASMs and Sequential Composition

Aggregation as we have described it thus far gives semantically acceptable results with Basic ASMs. However for Turbo ASMs, which allow for sequential composition and iteration of ASMs within one single step of the machine, this is insufficient. With the introduction of incremental updates resulting in the modification of elements at a given location, it is not always desirable for a Turbo ASM rule to return aggregated resultant updates (see Section 2.7.2).

In this subsection we discuss how support for sequential composition of update multisets is incorporated into the engine an extensible fashion. We then provide a modified version of the CoreASM *seq*-rule introduced in Specification 4.3 which uses both aggregation and sequential composition to produce its updates.

5.2.1 Update Multiset Composition

Recall that in our introduction to Gurevich's partial updates, we discussed their intricacies in Turbo ASMs and redefined sequential composition to accommodate them (see description of the $\tilde{\oplus}$ operator in Section 2.7.2). Similarly, with incremental updates we must provide a means of deriving the sequential composition of two update multisets. As such, we redefine sequential composition as expressed by the $\tilde{\oplus}$ operator in CoreASM

$$\tilde{\Delta}_{seq} = \tilde{\Delta}_i \tilde{\oplus} \tilde{\Delta}_{i+1} = \begin{cases} \text{Compose}(\tilde{\Delta}_i, \tilde{\Delta}_{i+1}), & \text{if } \text{consistent}(\tilde{\Delta}_i) \\ \tilde{\Delta}_i, & \text{otherwise.} \end{cases}$$

where $\tilde{\Delta}_i$ and $\tilde{\Delta}_{i+1}$ are update multisets produced by consecutive steps, $\tilde{\Delta}_{seq}$ is the update multiset resulting from the sequential composition of two others, and *consistent* returns true if the update multiset is consistent with respect to both aggregation and typical ASM

consistency conditions. It is important to note that the `Compose` rule expects both the aggregation of the first update multiset and the update set resulting from the aggregation of the first update multiset, to be consistent.

Notice that the `Compose` rule is essentially a black box. To support Turbo ASM semantics along with extensibility, all aggregator plug-ins are required to provide an algorithm which, when given two update multisets, will produce a multiset containing composed update instructions for all locations for which it is responsible. The semantics of sequential composition of incremental updates for which plug-ins are responsible are provided solely by them. A plug-in is deemed responsible for the composition of updates at a given location, iff one of the following holds:

- The plug-in is responsible for aggregation of the location based on the second update multiset.
- The plug-in is responsible for aggregation of a location based on the first update multiset, iff that location is not modified by the second update multiset.

$\widetilde{\Delta}_{seq}$ is the union of all composed update instructions produced by individual plug-ins.

Abstract Storage

```

Compose( $uMset_1$  : UPDATEMULTISET,  $uMset_2$  : UPDATEMULTISET)  $\equiv$ 
  let  $ap = \{a \mid a \in \text{PLUGIN} \wedge \text{aggregator}(a)\}$  in
    forall  $p \in ap$  do
      let  $R = \text{composerRule}(p)$  in
         $\text{composedUpdates}(p, uMset_1, uMset_2) \leftarrow R(uMset_1, uMset_2)$ 
    seq
    // Results in an update multiset
    result :=  $\bigcup_{p \in ap} \text{composedUpdates}(p, uMset_1, uMset_2)$ 

```

The *composedUpdates* function is used to collect the updates resulting from plug-ins performing sequential composition of two update multisets. The *composerRule* function is expected to return the rule from the given plug-in which implements the composition of updates on locations for which it is responsible. Note that the composition rule for each plug-in accepts two multisets as arguments, and its invocation results in the sequentially composed update multiset.

5.2.2 Basic Update Composer

To complement the Basic Update Aggregator, the Basic Update plug-in *buPlugin* also provides the *Basic Update Composer*. The Basic Update Composer is responsible for performing sequential composition of locations affected solely by basic updates. Recall (from Section 2.2) the \oplus operator for sequential composition of update sets in ASMs without partial or incremental updates. The composed update set produced by the \oplus operator, where Δ_1 is consistent an update set, is defined as:

$$\{\langle l, v \rangle \in \Delta_1 \mid l \notin \text{Locs}(\Delta_2)\} \cup \Delta_2$$

The Basic Update Composer processes all basic updates in update multisets for which it is responsible, in a similar fashion:

$$\{ui_1 \in \check{\Delta}_1 \mid uiLoc(ui_1) \notin \text{Locs}(\check{\Delta}_2) \wedge isBasicUpdate(ui_1)\} \cup \{ui_2 \in \check{\Delta}_2 \wedge isBasicUpdate(ui_2)\}$$

where $\text{Locs}(\check{\Delta}_2) = \{uiLoc(ui) \mid ui \in \check{\Delta}_2\}$. The addition of the *isBasicUpdate* function ensures that, in this more complicated case where update instructions are present, the algorithm handles only the sequential composition of those update instructions for which it is responsible. The Basic Update Composer is defined as follows:

Abstract Storage

BasicUpdateComposer(*uMset*₁ : UPDATEMULTISET, *uMset*₂ : UPDATEMULTISET) \equiv

```

result := {}
seq
  // {ui1 ∈ Δ1 | uiLoc(ui1) ∉ Locs(Δ2) ∧ isBasicUpdate(ui1)}
  forall ui1 ∈ uMset1 with ¬locUpdated(uMset2, uiLoc(ui1)) ∧
    isBasicUpdate(uMset1, ui1) do
    add ui1 to result
  // {ui2 ∈ Δ2 ∧ isBasicUpdate(ui2)}
  forall ui2 ∈ uMset2 with isBasicUpdate(uMset2, ui2) do
    add ui2 to result

```

where

$$isBasicUpdate(uMset, ui) \equiv \forall \langle l, v, a \rangle \in uMset, l = uiLoc(ui) \wedge a = updateAction$$

$$locUpdated(uMset, l) \equiv \exists ui \in uMset, l = uiLoc(ui)$$

The evaluation of *composerRule(buPlugin)* results in the rule **BasicUpdateComposer**; hence, the Basic Update Composer is called by **Compose** alongside all the sequential composition

algorithms provided by other plug-ins.

5.2.3 Interpretation of the Seq-rule

With the incorporation of aggregation and incremental updates into CoreASM, the interpretation of Turbo ASM rules is handled differently. The creation of temporary states upon evaluation of sequential and iterative rules requires the use of the **Aggregate** and **Compose** rules. We redefine the interpretation of the **seq**-rule introduced in Specification 4.3 to show how the introduction of aggregation into an ASM step affects Turbo ASM rules which themselves simulate ASM steps.

Before consistency of the update instructions produced by the first rule can be checked, aggregation of the resultant update multiset must be done. If both aggregation consistency and update set consistency hold, the resultant update set is applied to the current state producing a temporary state; otherwise the first update multiset is returned. If the update instructions produced by the first rule are consistent, the second rule is fired in the temporary state, resulting in the second update multiset. The first and second update multisets must then be sequentially composed. The update multiset resulting from sequential composition is the update multiset produced by the **seq**-rule in the simulated machine.

	Seq Rule with Aggregation
$(\alpha \boxed{r_1} \text{ seq } \beta \boxed{r_2}) \rightarrow$	$pos := \alpha$
$(\alpha u_1 \text{ seq } \beta \boxed{r_2}) \rightarrow$	local $uSet$ $uSet \leftarrow \text{Aggregate}(u_1)$ seq if $isConsistent(uSet) \wedge aggregationConsistent(u_1)$ then PushState Apply($uSet$) $pos := \beta$ else $[[pos]] := (undef, u_1, undef)$
$(\alpha u_1 \text{ seq } \beta u_2) \rightarrow$	local $uMset$ $uMset \leftarrow \text{Compose}(u_1, u_2)$ seq PopState $[[pos]] := (undef, uMset, undef)$

5.3 Set Plug-in

In this section we showcase the extensibility of CoreASM by providing the entire specification of the *Set Plug-in*. It provides all that is needed to work with finite sets as elements in the engine, including facilities for aggregation and incremental updates. The Set Plug-in extends the CoreASM engine with:

- The background class of sets. This is accomplished via:
 - An extension to Abstract Storage providing encoding and decoding functions for sets.
 - Extensions to the Parser and Interpreter defining concrete syntax and semantics for *set literals* including *set enumeration*, and *set comprehension*.
- Several set related operations extending the engine via:
 - An extension to the Parser and Interpreter defining concrete syntax and semantics for *set union*, *set intersection*, and *set difference* operators.
- The definition of set specific rule forms extending the engine via:
 - Extensions to the Parser and to the Interpreter defining concrete syntax and semantics for the **add-to** and **remove-from** rules which result in incremental updates to sets.
- Set specific aggregation and composition algorithms.

We will not discuss Parser extensions here (as this is discussed in more detail in Section 6.2). However, it suffices to say that Parser extensions involve the addition of productions to the CoreASM language grammar, and procedures to create the appropriate AST upon reduction of these productions.

Note that in the underlying machine, the Set Plug-in is represented by *setPlugin*, where $setPlugin \in \text{PLUGIN}$.

5.3.1 Background Extension

In the underlying machine, all set elements come from the SETELEMENT domain. The background of set elements is provided by *setBack*, where:

- $setBack \in \text{BACKGROUND}$
The set background is one of the backgrounds provided by the simulated machine.
- $name(setBack) = \text{"Set"}$
The name of the set background is "Set".
- $newValue(setBack) = \text{a SETELEMENT representing the empty set}$
When a new value is of the set background is required, a set element containing no members is returned.

Set Membership

All set elements in the simulated machine belong to the set background:

- $\forall s \in \text{SETELEMENT}, bkg(s) = \text{"Set"}$.

The set background provides an interface with the functionality required to represent and access information about the internal structure of set elements in the simulated machine:

- **controlled** $setMember : \text{SETELEMENT} \times \text{ELEMENT} \rightarrow \text{BOOLEAN}$
holds *true* if the element is a member of this set, and *false* otherwise.

Enumerability Of Input

An ASM may contain structures that group elements in different ways: universes, sets, multisets, trees. At times it is convenient to have one single view of all the various kinds of structures in an ASM which represent a group, so that these similar but different structures can be used in the same context by the CoreASM engine. As such, the kernel provides the *enumerability interface* which a background can implement for its elements. This interface is defined formally in Appendix A.1.2.

All set elements implement the enumerability interface as follows:

- $\forall s \in \text{SETELEMENT}, enumerable(s)$
All set elements are enumerable.
- **derived** $enumerateset : \text{SETELEMENT} \rightarrow \text{ELEMENTCOLLECTION}$
The enumeration of a set, provides a collection⁶ containing all elements of the set:

⁶Notice our use of square braces to denote a collection. We define a collection as a simple group of items with no axioms imposed on its structure.

$$\text{enumerate}(s) \equiv [e \mid e \in \text{ELEMENT} \wedge \text{setMember}(s, e)]$$

Set Literals

The Set Plug-in provides two methods of set description: namely set enumeration and set comprehension. Set enumeration affords the description of the members of a set by individually enumerating each of the elements it contains:

	Set Enumeration
$\langle \{ \lambda_1 \text{?}_1, \dots, \lambda_n \text{?}_n \} \rangle \rightarrow$	<pre> choose $i \in [1..n]$ with $\neg \text{evaluated}(\lambda_i)$ $pos := \lambda_i$ ifnone let $newSet = \text{newValue}(\text{setBack})$ in forall $i \in [1..n]$ $\text{setMember}(newSet, \text{value}(\lambda_i)) := \text{true}$ $[[pos]] := (\text{undef}, \text{undef}, newSet)$ </pre>

Set comprehension allows one to describe set contents algorithmically. Since set comprehension was born in the mathematical world rather than the computational world, there are many accepted syntactic and semantic variants of it. Given the general set comprehension expression form

$$\{x_0 \text{ is } exp_0 \mid x_1 \text{ in } exp_1, \dots, x_n \text{ in } exp_n \text{ with } exp_g\}$$

we refer to the free variable x_0 as the *specifier variable*, the expression exp_0 as the *specifier expression*, the free variables $x_1 \dots x_n$ as the *constrainer variables*, $exp_1 \dots exp_n$ as the *constrainer expressions*, and exp_g as the *guard*. In CoreASM we provide two variants which encompass a wide range of algorithmically expressible finite sets. While the PA rules for variants presented contain guards, both variants can be used without a guard (which is equivalent to the guard being set to tt).

The simplest variant of set comprehension binds the specifier variable to each element in the constrainer expression, adding any element which satisfies the guard to the resulting set. Note that within the guard, the specifier variable should be a free variable:

```

( $\{ \alpha x \mid \beta x_1 \text{ in } \gamma \varrho \text{ with } \delta \varrho \}$ )  $\rightarrow$ 
  if  $x = x_1$  then
     $pos := \gamma$ 
     $considered(pos) := \{ \}$ 
     $newSet(pos) := newValue(setBack)$ 
  else
    Error('Constrainer variable must have same name as specifier variable')

( $\{ \alpha x \mid \beta x_1 \text{ in } \gamma v \text{ with } \delta \varrho \}$ )  $\rightarrow$ 
  if  $\neg enumerable(value(\gamma))$  then
    Error('Free variables may only be bound to enumerable elements')
  else
    let  $s = enumerate(value(\gamma)) \setminus considered(pos)$  in
      choose  $e \in s$  do
         $env(x) := e$ 
        add  $e$  to  $considered(pos)$ 
         $pos := \delta$ 
      ifnone
         $[[pos]] := (undef, undef, newSet(pos))$ 

( $\{ \alpha x \mid \beta x_1 \text{ in } \gamma v \text{ with } \delta v \}$ )  $\rightarrow$ 
  if  $value(\delta) = tt$  then
     $setMember(newSet(pos), env(x)) := true$ 
     $env(x) := undef$ 
    ClearTree( $\delta$ )
     $pos := \delta$ 

```

The *considered* function is used to keep a record of all elements of the constrainer expression which have already been considered for addition into the resultant set. This variant supports set comprehension expressions such as:

$$\{x \mid x \text{ in } aUniverse\}$$

$$\{x \mid x \text{ in } 1..100 \text{ with } mod(x, 2) = 0\}$$

Note that this variant is particularly useful for creating set elements from other enumerable elements.

The second variant allows the specifier to be defined in terms of the specifier expression; the constrainer variables are themselves expected to be present in the specifier expression, and this expression is re-evaluated for all possible combinations of the constrainer variables. If the guard is satisfied, the result of the evaluation of the specifier expression is added to the resultant set:

Set Comprehension : Variant B

```

( $\{ \alpha x \text{ is } \epsilon[e] \mid \beta_1 x_1 \text{ in } \gamma_1[\gamma]_1, \beta_2 x_2 \text{ in } \gamma_2[\gamma]_2, \dots, \beta_n x_n \text{ in } \gamma_n[\gamma]_n \text{ with } \delta[\gamma] \}$ )  $\rightarrow$ 
  if gtOneConstVariable then
    if notSameNameConstSpecVar then
      choose  $j \in [1..n]$  with  $value(\gamma_j) = undef$  do
         $pos := \gamma_j$ 
      ifnone
        if sameNameTwoConstVar then
          Error('No two constrainer variables may have the same name')
        else if constExpNotEnumerable then
          Error('Constrainer variables may only be bound to enumerable elements')
        else if constExpEmptyEnumerable then
           $[[pos]] := (undef, undef, newValue(setBack))$ 
        else
           $newSet(pos) := newValue(setBack)$ 
          InitializeChooseConsideredCombos
           $pos := \delta$ 
    else
      Error('Constrainer variable cannot have same name as specifier')
  else
    Error('At least one constrainer variable must be present')
where
  gtOneConstVariable  $\equiv n \geq 1$ 
  notSameNameConstSpecVar  $\equiv \exists j \in [1..n], x \neq x_j$ 
  sameNameTwoConstVar  $\equiv \exists k \in [1..n], \exists l \in [1..n] \ k \neq l \wedge x_k = x_l$ 
  constExpNotEnumerable  $\equiv \exists m \in [1..n], \neg enumerable(value(\gamma_m))$ 
  constExpEmptyEnumerable  $\equiv \exists p \in [1..n] \ |enumerate(value(\gamma_p))| = 0$ 

```

```

( $\{ \alpha x \text{ is } \epsilon \square \mid \beta_1 x_1 \text{ in } \gamma_1 v_1, \beta_2 x_2 \text{ in } \gamma_2 v_2, \dots, \beta_n x_n \text{ in } \gamma_n v_n \text{ with } \delta v \}$ )  $\rightarrow$ 
  if  $value(\delta) := tt$  then
     $pos := \epsilon$ 
  else
    if OtherCombosToConsider then
      ChooseNextCombo
      ClearTree( $\delta$ )
       $pos := \delta$ 
    else
      ClearConsideredCombos
       $[[pos]] := (undef, undef, newSet(pos))$ 

( $\{ \alpha x \text{ is } \epsilon v \mid \beta_1 x_1 \text{ in } \gamma_1 v_1, \beta_2 x_2 \text{ in } \gamma_2 v_2, \dots, \beta_n x_n \text{ in } \gamma_n v_n \text{ with } \delta v \}$ )  $\rightarrow$ 
   $setMember(newSet(pos), value(\epsilon))$ 
  if OtherCombosToConsider then
    ChooseNextCombo
    ClearTree( $\delta$ )
    ClearTree( $\epsilon$ )
     $pos := \delta$ 
  else
    ClearConsideredCombos
     $[[pos]] := (undef, undef, newSet(pos))$ 

```

The macros InitializeChooseConsideredCombos, ChooseNextCombo, ClearConsideredCombos and OtherCombosToConsider are used to consider every possible combination of the elements from the n constrainer expressions. The actual assignment of elements to constrainer variables is done within these rules. A formal definition of each of these rules is provided in Appendix A.2.

Variant B is quite a bit more expressive than variant A, as it allows sets consisting of more complex elements to be built. For example:

$$\{x \text{ is } y * z \mid y \text{ in } \{1, 3, 5\}, z \text{ in } \{2, 4, 6\}\}$$

$$\{x \text{ is } \{a, b, c\} \mid a \text{ in } 1..100, b \text{ in } \{1, 2, 3\}, c \text{ in } aSet\}$$

Notice that in the second example this variant is used to build a set of sets.

5.3.2 Operators Extension

The Set Plug-in provides the set union, set intersection, and set difference operators along with a definition for equality of set elements.

Operators

You will notice that all the set operators provided by the Set Plug-in can operate on any enumerable element. However, the result produced by all operators is always a set element. The set intersection operator results in a set element containing the intersection of its operands:

Set Intersection

```

( $\alpha$ [?]  $\cap$   $\beta$ [?]) → choose  $\lambda \in \{\alpha, \beta\}$  with  $\neg$ evaluated( $\lambda$ )
                         $pos := \lambda$ 
ifnone
  if enumerable(value( $\alpha$ ))  $\wedge$  enumerable(value( $\beta$ )) then
    let newSet = newValue(setBack) in
      forall  $eL \in$  enumerate(value( $\alpha$ )) do
        choose  $eR \in$  enumerate(value( $\beta$ )) with equal( $eR, eL$ ) do
          setMember(newSet,  $eL$ ) := true
         $[[pos]] := (undef, undef, newSet)$ 
      else
        Error('Both operands must be enumerable.')

```

The set difference operator results in a set element containing all elements in the LHS operand which are not in the RHS operand:

Set Difference

```

( $\alpha \setminus \beta$ ) → choose  $\lambda \in \{\alpha, \beta\}$  with  $\neg \text{evaluated}(\lambda)$ 
                pos :=  $\lambda$ 
            ifnone
            if  $\text{enumerable}(\text{value}(\alpha)) \wedge \text{enumerable}(\text{value}(\beta))$  then
                let newSet = newValue(setBack) in
                forall  $eL \in \text{enumerate}(\text{value}(\alpha))$  do
                    if  $\nexists eR \in \text{enumerate}(\text{value}(\beta)), \text{equal}(eR, eL)$  then
                        setMember(newSet, eL) := true
                    [[pos]] := (undef, undef, newSet)
                else
                    Error('Both operands must be enumerable.')

```

The set union operator results in a set element containing all elements contained in both operands:

Set Union

```

( $\alpha \cup \beta$ ) → choose  $\lambda \in \{\alpha, \beta\}$  with  $\neg \text{evaluated}(\lambda)$ 
                pos :=  $\lambda$ 
            ifnone
            if  $\text{enumerable}(\text{value}(\alpha)) \wedge \text{enumerable}(\text{value}(\beta))$  then
                let newSet = newValue(setBack) in
                forall  $eL \in \text{enumerate}(\text{value}(\alpha))$  do
                    setMember(newSet, eL) := true
                forall  $eR \in \text{enumerate}(\text{value}(\beta))$  do
                    setMember(newSet, eR) := true
                [[pos]] := (undef, undef, newSet)
            else
                Error('Both operands must be enumerable.')

```

Equivalence Definition

Notice that all operations provided result in a new set element being created and manipulated, rather than the modification of an existing set element. This implies that it is possible to have two unique set elements in a state of the simulated machine which both represent the same set. As such, uniqueness of set elements in the simulated machine is not enough to determine the equivalence of two set elements. Recall that the equivalence operator shown

in Specification 4.4 relies on the background to provide its own notion of equality of its own elements; for set elements, equality is formally defined as follows:

- **derived** $equal_{Set} : \text{ELEMENT} \times \text{ELEMENT} \rightarrow \text{BOOLEAN}$

where we have:

$$\begin{aligned} equal_{Set}(s_1, s_2) \equiv & \\ & s_1, s_2 \in \text{SETELEMENT} \\ & \wedge \forall e_1 \in enumerate(s_1), \exists e_2 \in enumerate(s_2), equal(e_1, e_2) \\ & \wedge |enumerate(s_1)| = |enumerate(s_2)| \end{aligned}$$

For two set elements to be equal, they must represent sets containing the same elements and be the same size. Notice that the CoreASM equality function $equal$ (defined in Appendix Section A.1.1) is used to ensure that member elements of s_1 are also members of s_2 .

5.3.3 Rule Extension

To facilitate incremental updates to sets, the **add-to-rule** and **remove-from-rule** are supported by the Set Plug-in. The addition of an element to a set using the **add-to-rule** results in an update instruction consisting of the $setAddAction$ action. It is formally defined as

$$\begin{aligned} (\text{add } \alpha[e] \text{ to } \beta[e]) \rightarrow & \text{choose } \tau \in \{\alpha, \beta\} \text{ with } value(\tau) = undef \\ & pos := \tau \\ & \text{ifnone} \\ & \llbracket pos \rrbracket := (undef, \{\langle loc(\beta), value(\alpha), setAddAction \rangle\}, undef) \end{aligned}$$

Add-To Rule

where $setAddAction \in \text{ACTION}$.

The removal of a single element from a set, or the **remove-from-rule**, results in an update instruction containing a $setRemoveAction$ action. Its formal definition is

$$\begin{aligned} (\text{remove } \alpha[e] \text{ from } \beta[e]) \rightarrow & \text{choose } \tau \in \{\alpha, \beta\} \text{ with } value(\tau) = undef \\ & pos := \tau \\ & \text{ifnone} \\ & \llbracket pos \rrbracket := (undef, \{\langle loc(\beta), value(\alpha), setRemoveAction \rangle\}, undef) \end{aligned}$$

Remove-From Rule

where $setRemoveAction \in ACTION$.

Notice that in both rules, no checks are made to ensure that a set is being manipulated. This check is deferred to when sets are aggregated.

5.3.4 Aggregation Algorithm

The aggregation algorithm described here facilitates the aggregation of instructions produced by the **add-to**, **remove-from**, and **update**-rules; successful aggregation produces a resultant set element for each aggregated location. Note that our approach to incremental modification of sets is similar in spirit to both the set integration example in [38] and the AsmL specific approach briefly discussed in [37], and achieves the same results.

It is worthwhile to note that the Set Plug-in is only required to aggregate update instructions operating on those locations for which the Set Plug-in is responsible. Put another way, the aggregation algorithm for sets will not aggregate update instructions for a location unless there is a set-related incremental update instruction which will modify that location. When only basic updates resulting in a set are made to a given location, the Basic Update Aggregator introduced in Section 5.1.5 takes care of their aggregation.

When the aggregation phase of the machine is being executed, $aggregatorRule(setPlugin)$ results in the rule `AggregateSets`:

```

AggregateSets(uMset : UPDATEMULTISET) ≡
  local resultantUpdate
    result := {}
  seq
    forall l ∈ locsToAggregate do
      if regularUpdatesExist then
        // Case 1a
        if inconsistentRegularUpdates then
          HandleInconsistentAggregation(l, uMset, setPlugin)
        // Case 1b
        else if regularUpdateIsNotSet
          HandleInconsistentAggregation(l, uMset, setPlugin)
        // Case 1c
        else if addRemoveConflictWithRU then
          HandleInconsistentAggregation(l, uMset, setPlugin)
        else
          resultantUpdate ← GetRegularUpdate(l, uMset)
          seq
            add resultantUpdate to result
      else
        // Case 2a
        if addRemoveConflict then
          HandleInconsistentAggregation(l, uMset, setPlugin)
        // Case 2b
        else if setNotInLocation then
          HandleInconsistentAggregation(l, uMset, setPlugin)
        else
          resultantUpdate ← BuildResultantUpdate(l, uMset)
          seq
            add resultantUpdate to result

```

where

$$\begin{aligned}
\text{locsToAggregate} &\equiv \{l \mid \langle l, v, a \rangle \in uMset \wedge a \in \{\text{setAddAction}, \text{setRemoveAction}\}\} \\
\text{regularUpdatesExist} &\equiv \exists \langle l, v, \text{updateAction} \rangle \in uMset \\
\text{inconsistentRegularUpdates} &\equiv \exists \langle l, v_1, \text{updateAction} \rangle \in uMset, \exists \langle l, v_2, \text{updateAction} \rangle \in uMset, \\
&v_1 \neq v_2 \\
\text{regularUpdateIsNotASet} &\equiv \exists \langle l, v, \text{updateAction} \rangle \in uMset, \text{bkg}(v) \neq \text{"Set"} \\
\text{addRemoveConflictWithRU} &\equiv \text{addConflictWithRU} \vee \text{removeConflictWithRU} \\
\text{addConflictWithRU} &\equiv \exists \langle l, vRU, \text{updateAction} \rangle \in uMset, \exists \langle l, vSU, \text{setAddAction} \rangle, \\
&vSU \notin \text{enumerate}(vRU) \\
\text{removeConflictWithRU} &\equiv \exists \langle l, vRU, \text{updateAction} \rangle \in uMset, \exists \langle l, vSU, \text{setRemoveAction} \rangle, \\
&vSU \in \text{enumerate}(vRU) \\
\text{addRemoveConflict} &\equiv \exists \langle l, v, \text{setAddAction} \rangle \in uMset, \exists \langle l, v, \text{setRemoveAction} \rangle \in uMset \\
\text{setNotInLocation} &\equiv \text{bkg}(\text{GetValue}(l)) \neq \text{"Set"}
\end{aligned}$$

Set aggregation is done on a per-location basis. It is important to note that set aggregation of a location consisting only of a number of incremental updates is handled differently than the case where both incremental updates and regular updates are aggregated. This is the case with both inconsistency checking and the actual procedure of aggregation.

All consistency checks for a location are performed before the actual aggregation takes place. Violation of any one of the following requirements results in the aggregation of the location being considered inconsistent (comments appearing in the formal definition of set aggregation correspond to inconsistency cases described here):

1. If there is a regular update to a given location along with incremental updates:
 - (a) There cannot exist two regular updates to the location, resulting in two unique values; this is a typical consistency check of regular updates.
 - (b) All regular updates to a location may only result in a set element.
 - (c) There cannot be a *setAddAction* of an element not found in the set value of a regular update, nor can there be a *setRemoveAction* of an element found in the set value of a regular update to the location.
2. If there are only incremental updates to a given location
 - (a) There cannot exist two incremental updates, one performing a *setAddAction* and the other performing a *setRemoveAction*, on the same location and for the same

value.

- (b) The value of the location in the current state must be a set element.

When set incremental updates and set regular updates are successfully aggregated, the resultant update is simply the value of (any of) the regular updates to the location; all regular updates to the location should be consistent, and all incremental updates should not conflict with the regular update value:

GetRegularUpdate

```

GetRegularUpdate( $l : \text{LOC}, uMset : \text{UPDATEMULTISET}$ )  $\equiv$ 
  forall  $ui \in uMset$  with  $uiLoc(ui) = l$  do
    if  $uiAction(ui) = updateAction$ 
      result :=  $ui$ 
   $aggStatus(ui, setPlugin) := successful$ 

```

The keen observer may notice that the **result** function is potentially assigned a value within each thread of the **forall**-rule, and that if **result** is updated to two unique values there would be an inconsistency in the underlying machine. However the aggregation consistency checking done before **GetRegularUpdate** is called, and in particular the check done with the *inconsistentRegularUpdates* derived function, ensures that if multiple regular updates to the given location do exist, that they are identical as well.

The resultant update produced by the successful aggregation of incremental updates alone is the set resulting from the addition and removal of elements from the set at the location in the current state.

BuildResultantUpdate

```

BuildResultantUpdate( $l : \text{LOC}, uMset : \text{UPDATEMULTISET}$ )  $\equiv$ 
  local  $newSet := newValue(setBack)$ 
  forall  $e \in enumerate(GetValue(l))$ 
    if  $\exists(l, e, setRemoveAction)$  do
       $setMember(newSet, e) := true$ 
  forall  $\langle l, v, setAddAction \rangle$  do
     $setMember(newSet, v) := true$ 
  result :=  $\langle l, newSet, updateAction \rangle$ 
  forall  $ui \in uMset$  with  $uiLoc(ui) = l$  do
     $aggStatus(ui, setPlugin) := successful$ 

```

5.3.5 Composition Algorithm

To support aggregation in Turbo ASMs, the Set Plug-in provides a sequential composition algorithm. When the sequential composition of two update multisets is requested, $composerRule(setPlugin)$ results in the rule `ComposeSets`:

`ComposeSets`

```

ComposeSets( $uMset_1$  : UPDATEMULTISET,  $uMset_2$  : UPDATEMULTISET)  $\equiv$ 
  result := {}
  seq
  forall  $l \in locsAffected$  do
    // Case 1a
    if  $locAddRemove(uMset_1) \wedge \neg locUpdated(uMset_2)$  then
      forall  $ui \in uMset_1$  with  $uiLoc(ui) = l$  do
        add  $ui$  to result
    // Case 1b
    else if  $\neg locUpdated(uMset_1) \wedge locAddRemove(uMset_2)$  then
      forall  $ui \in uMset_2$  with  $uiLoc(ui) = l$  do
        add  $ui$  to result
    // Case 2
    else if  $locAddRemove(uMset_2) \wedge locRegularUpdate(uMset_2)$  then
      forall  $ui \in uMset_2$  with  $uiLoc(ui) = l$  do
        add  $ui$  to result
    // Case 3a
    else if  $locAddRemove(uMset_1) \wedge locRegularUpdate(uMset_1) \wedge locAddRemove(uMset_2)$  then
      add AggregateLocation( $l, uMset_1, uMset_2$ ) to result
    // Case 3b
    else if  $locAddRemove(uMset_1) \wedge locAddRemove(uMset_2)$  then
      forall  $ui \in EradicateConflictingIncrementalUpdates(l, uMset_1, uMset_2)$  do
        add  $ui$  to result
  where
     $locsAffected \equiv \{l_1 \mid \langle l, v, a \rangle \in uMset_1\} \cup \{l_2 \mid \langle l, v, a \rangle \in uMset_2\}$ 
     $locAddRemove(uMset) \equiv \exists \langle l, v, a \rangle \in uMset, a = setAddAction \wedge a = setRemoveAction$ 
     $locRegularUpdate(uMset) \equiv \exists \langle l, v, a \rangle \in uMset, a = updateAction$ 
     $locUpdated(uMset) \equiv \exists \langle l, v, a \rangle \in uMset$ 

```

The Set Plug-in will only perform sequential composition on locations for which it is responsible. This plug-in is responsible for composition of a given location when that location

is affected by set-related incremental update instructions:

- In the second update multiset.
- In the first update multiset, and the location is not affected at all by any update in the second update multiset.

When only basic updates resulting in a set are made to a given location, the Basic Update Composer introduced in Section 5.1.5 takes care of their sequential composition.

Comments appearing in the formal definition of set aggregation correspond to the cases where set composition is the responsibility of the plug-in, as described here:

1. Include update instructions for locations affected exclusively by only one of the steps:
 - (a) Include update instructions for locations in the first step, which are not updated in the second step.
 - (b) Include all update instructions for locations from the second step, if those locations are not updated in the first step.
2. If a regular update along with incremental updates affect a location in the second step, include all update instructions from only the second step, excluding all from the first.
3. Manipulate update instructions on a single location that if simply added (without any transformation) from the first and second step into the resultant update multiset would conflict with each other causing inconsistency:
 - (a) A regular update which occurred in the first step (e.g. $\langle l, \{1, 2\}, updateAction \rangle$), may conflict with incremental updates which occurred in the second step (e.g. $\langle l, 2, setRemoveAction \rangle$) if included together in the same incremental updates multiset. In this situation, sequential composition should produce a regular update. The regular update produced is created by aggregating the incremental updates of the second step, with the assumption that the location currently contains the value of the regular update from the first step.

AggregateLocation

AggregateLocation($l : \text{LOC}, uMset_1 : \text{UPDATEMULTISET},$
 $uMset_2 : \text{UPDATEMULTISET}) \equiv$

return *resultantUpdate* **in**

let *newSet* := *newValue(setBack)* **do**

forall $e \in \text{enumerate}(\text{getLocRUValue}(uMset_1))$

if $\exists \langle l, e, \text{setRemoveAction} \rangle \in uMset_2$ **do**

setMember(newSet, e) := true

forall $\langle l, v, \text{setAddAction} \rangle \in uMset_2$ **do**

setMember(newSet, v) := true

resultantUpdate := \langle l, newSet, updateAction \rangle

where

getLocRUValue($uMset$) $\equiv v$ s.t. $\langle l, v, a \rangle \in uMset \wedge a = \text{updateAction}$

- (b) Two instructions performing a *setAddAction* and *setRemoveAction* to the same value causes aggregation consistency failure. However, such instructions occurring in sequence nullify one another:
- i. For any location, a *setAddAction* occurring in the first step followed by a *setRemoveAction* in the second, clearly causes no change to a given set upon the completion of the second step. Update instructions containing both these opposing actions on the same location should be omitted from the final composed update multiset.
 - ii. For any location on which a *setRemoveAction* in the first step is followed by a *setAddAction* in the second, the removal is nullified by the addition. Thus, update instructions containing such a *setRemoveAction* should be excluded from the composed update multiset.
 - iii. All other updates operating on a location should be included in the composition of the location.

```

EradicateConflictingIncrementalUpdates
EradicateConflictingIncrementalUpdates( $l : \text{LOC}, uMset_1 : \text{UPDATEMULTISET},$ 
 $uMset_2 : \text{UPDATEMULTISET}) \equiv$ 

return remainingUpdates in
  remainingUpdates := {}
  seq
    forall  $v \in \text{locValues}$  do
      // Case 3(b)i
      if  $\text{locValAct}(uMset_1, v, \text{setAddAction}) \wedge$ 
 $\text{locValAct}(uMset_2, v, \text{setRemoveAction})$  then
        skip
      // Case 3(b)ii
      else if  $\text{locValAct}(uMset_1, v, \text{setRemoveAction}) \wedge$ 
 $\text{locValAct}(uMset_2, v, \text{setAddAction})$  then
        forall  $ui \in \{\langle l, v, \text{setAddAction} \rangle \in uMset_2\}$  do
          add  $ui$  to remainingUpdates
        // Case 3(b)iii
      else
        forall  $ui \in \text{getAllLocValUpdates}$  do
          add  $ui$  to remainingUpdates
  where
     $\text{locValues} \equiv \{v_1 \mid \langle l, v_1, a_1 \rangle \in uMset_1\} \cup \{v_2 \mid \langle l, v_2, a_2 \rangle \in uMset_2\}$ 
     $\text{locValAct}(uMset, v, a) \equiv \exists \langle l, v, a \rangle \in uMset$ 
     $\text{getAllLocValUpdates} \equiv \{\langle l, v, a_1 \rangle \in uMset_1\} \cup \{\langle l, v, a_2 \rangle \in uMset_2\}$ 

```

5.4 Summary

In this chapter we described how distributed incremental change is facilitated in the CoreASM simulated machine using our framework of incremental updates and aggregation. We use the Set Plug-in as an example of general engine extensibility. Our approach offloads the burden of aggregation and sequential composition of particular sorts to the appropriate plug-in. It is important to note that the semantics of aggregation and sequential composition are unknown to the kernel, and in the hands of the plug-ins responsible. This promotes unlimited future extensibility of CoreASM with background classes of aggregatable sorts.

“Call now. Operators are standing by.”

— Every late-night infomercial

Chapter 6

Operator Evaluation and Language Additions

Recall that in our introductory discussion of ASMs, while syntax and semantics of rule forms were presented, we do not mention specific sorts or operators provided by ASMs. This is due to the fact that ASMs do not restrict the user to any predetermined data types¹ or operators², thus allowing for extreme flexibility in modelling complex systems. Also recall that many syntactic and semantic extensions have been introduced beyond Basic ASMs, including Turbo ASMs and incremental modification of elements. It is not unusual for such conveniences to be introduced and used in a specific model, provided that their semantics are well-defined and that they operate within the semantic constraints of the ASM paradigm. In this chapter, we discuss some of the specific CoreASM engine design decisions made to facilitate such freedom and flexibility of modelling.

We will first discuss issues which arise during operator evaluation in the CoreASM engine. We then describe how these issues are addressed. This is followed by a discussion of general language extensibility and how this is accomplished. In particular, we describe how the parser used by the Parser module, and the grammar accepted by the parser, is extended by plug-ins and generated dynamically based on individual specification requirements.

¹The only required data type is Boolean.

²The only required operator is the equals sign “=” for equality.

6.1 Operator Extension and Evaluation

In Section 5.3 we gave a formal definition of the Set Plug-In, which provides well-defined syntax and semantics for the set sort in CoreASM. Along with the definition of this sort came several definitions of operations on set elements (i.e. set union, set difference, and set intersection). In general any plug-in may extend the CoreASM language with new operators. Here we formally specify the procedure used to evaluate operators in CoreASM.

6.1.1 ASM Type Conventions

ASM typing is a nebulous topic that deserves a lengthy discussion that is beyond the scope of this thesis. We give it only a cursory coverage here, since it is important only to our discussion of the significance of operator evaluation in CoreASM. We shall present a view of ASM typing which is convenient for our purposes and does not contradict the well-defined semantics of ASMs.

ASMs by definition have no concrete typing conventions³. However the notion of a type, or in algebraic specification language terms a *sort*, does exist; as mentioned in Section 2.1, universe membership allows any state of an ASM to be viewed as a many-sorted structure. Elements introduced by background classes are identified using universe membership. For example the Boolean background class provides the universe `BOOLEAN` containing the elements *true* and *false*. It is important to note that the sorts available for use in a specification are not dependent on what ASMs provide by default. Rather, the sorts available for use are those defined for use in a given ASM specification; sets, trees, numbers, strings and any other user defined data structure may be used if defined.

There is no restriction on how many universes an element may be a member of. This means that an element may be many sorts simultaneously. For instance, the *true* element may also be a member of the user-defined universe `TRUTH`; the *true* element is then of the sorts *truth*, *boolean* and *superuniverse*.

One may argue that ASMs are untyped because it seems like there is no predefined restriction on the kind of sort which is contained in a location or is used in a location tuple or `named-rule` call. We can associate this seeming lack of typing to *parametric polymorphism*: every location and rule parameter can be an element of the superuniverse

³The only type related restriction is that predicates must always result in a value of *true* or *false*.

(and by the definition of ASMs, every element is a member of the superuniverse). Another view that we can take is that all elements of an ASM have a common sort which we refer to here as the *base sort*; this implies that all elements are members of the *superuniverse*. In other words, all elements, regardless of the different universes they belong to, have a common sort. We can then attribute the untyped characteristic to the assumption that locations and rule parameters are always expected to be of the base sort. In any case, there is never a type inconsistency experienced with locations and rule parameters in ASMs.

Strictly-typed ASMs which require that all locations, location arguments, and rule arguments be bound to particular sorts have been proposed in [17, 54, 55, 56, 57]. Functions and rules must have *signatures* defined which express the expected sorts for each parameter. These specialized ASM variants have not, however, been fully embraced. The majority of the community feel that freedom-of-typing allows for rapid creation of high-level specifications by abstracting away from type specific details which are not important to the problem at hand. Only as specifications become more complex and begin to describe lower level intricacies does strict typing become more useful than restrictive.

6.1.2 Operators in ASMs

While ASMs seem untyped with respect to locations and rules, typing can, and often does, play a part in the evaluation of operators in ASMs. Operators used in a specification may either have generally accepted semantics (e.g. “+” for numerical addition) which require no further explanation, or have semantics that are explicitly defined for use in the specification. Nevertheless, to be used in a formal specification, operators must have well-defined semantics.

Operators with overloaded meanings can be useful. For example, the “+” operator can be defined to produce results dependent on the sorts of operands used (e.g. numerical addition if both operands are numbers, and string concatenation if both operands are strings):

$$e_1 + e_2 = \begin{cases} numAddition(e_1, e_2), & \text{if } e_1, e_2 \in \text{NUMBER} \\ stringConcat(e_1, e_2), & \text{if } e_1, e_2 \in \text{STRING} \end{cases}$$

6.1.3 CoreASM Type Conventions and Operator Extension

CoreASM type conventions are based on the classic definition of ASMs. All elements of the simulated state are of the (base) sort SuperUniverse. Sorts are defined via the introduction

of universes and their elements by the user with background plug-ins. Location arguments and values, along with **named**-rule call parameters, are expected to be elements of the base sort. As such, the use of locations and rules require no type checking.

Operators may be introduced by any plug-in. Recall that the syntax and semantics of each operator provided by the Set Plug-In is fully contained in PA rules; we refer to PA rules that together describe a behaviour of an operator as one *operator behaviour* (OB). An OB is a completely self-contained definition of the syntax and semantics of an operator in CoreASM. With the behaviour of an operator fully modularized, the engine can be extended with new operators without it needing to be aware of their syntax and semantics; to it, an OB is a black box which can be used to compute the result of operator evaluation. Like ASMs, CoreASM is not bound to any particular set of operators, but may be extended with an arbitrary number of operators.

CoreASM allows for the definition of overloaded operators by allowing multiple OBs to be introduced for a single operator, each potentially originating from a different plug-in; OBs for overloaded operators have a common syntax but differ in their semantics. The OBs which together define the overloaded behaviour of an operator are referred to collectively as an *operator definition* (OD). In an OD with overloaded OBs, the OBs are not aware of each other or their differing semantics. Because of this black box approach, the challenge of determining which (if any) OBs are appropriate for the given operands arises during operator evaluation.

The semantics of an OB may depend on its operands being of a specific sort. The OB may perform a type check to ensure that the operands are of a sort supported by the operation it provides. This means that typing becomes important in CoreASM during operator evaluation.

Because locations in the simulated state may hold any element, and operator type is only queried at interpret-time by OBs if they so choose, the CoreASM language is clearly dynamically typed⁴. However, we cannot categorize CoreASM as either weakly or strongly typed. Because plug-ins may introduce operators as well as background classes for any number of sorts, an OB can never be aware of all the sorts available for use. The plug-in

⁴There are plans to introduce **loose** and **strict** directives to CoreASM which will require function signatures to be defined *a priori*; upon function and rule definition, the sort expected for each argument and value will have to be defined. In these two cases either the user will, respectfully be warned of type inconsistency, or type consistency will be enforced by not allowing the ASM to run at all. Both of these options would allow for increasingly more strict typing conventions as specifications become more mature.

writer may provide OBs which are capable of sort coercion if they so choose. As a result, CoreASM is as strongly or weakly typed as plug-in writers make it.

6.1.4 Significance of Operator Evaluation in CoreASM

Because all CoreASM operators are defined using OBs and the language is dynamically typed, at interpret-time the engine must determine which OB (if any) applies to each usage of an operator in a specification.

Dynamically typed programming languages such as Python similarly allow for operator behaviour to be overloaded by the user⁵ (see [43, Chapter 21]). In such languages, at interpret-time the correct operator implementation is chosen based on matching the types of the operands with the types accepted by an operator implementation. Even though the CoreASM engine is completely unaware of the requirements of OBs, including the sorts of operands that they are able to operate on, it must determine which OB is the appropriate one to use in each case.

To accomplish this, the engine simply allows each OB to perform its own computation using the operands. It then decides if a valid result exists for the operation based on the examination of results produced by all OBs. If an operator completes without error, the result it returns is the result of its operation. We shall, in the following sections, formally define how operator evaluation works in CoreASM.

Clearly, both the complexity of this problem and the efficiency of operator evaluation could be reduced by requiring that OBs inform the engine of the sorts of elements they are able to operate on. However, making OBs explicitly type-dependent will unnecessarily stifle freedom of experimentation by forcing OBs to be designed with operand sort in mind. At the early stages of system design, the correctness of a model, which is facilitated through freedom of experimentation and exploration of the problem space, is more important than efficiency of interpretation.

6.1.5 CoreASM Operator Evaluation

When an OB is given operands to operate on, it is expected to return a result. Upon successful completion of the operation, the result is an element of the simulated machine's

⁵Python only allows for existing arithmetic operators to be overloaded. However, what is important to our discussion is method of deciding which of the overloaded operator implementations is used.

SuperUniverse, which is itself the underlying machine's ELEMENT domain. If an error, including one that is related to type checking, has occurred during the OB's attempt to complete the operation, it results in an element of the underlying machine's ERROR domain.

Recall that the ExecuteTree rule presented in Specification 4.1 implements the interpretation of AST nodes in the engine and is responsible for giving control of interpretation of a node to the appropriate plug-in. There the *plugin* function serves as an oracle which, when given a node, results in the plug-in that is responsible for the interpretation of the node. To facilitate operator evaluation, we refine the *plugin* function such that for all nodes representing operators, the *plugin* function does not need to behave as an oracle. Rather, it performs a procedure that, after determining the correct OB for the operator node, returns the appropriate plug-in which contains that OB:

- *plugin* : NODE \rightarrow PLUGIN is the plug-in associated with the node, and hence responsible for parsing and evaluating it. For nodes which are operators of the simulated specification, the DetermineOperatorPlugin rule is called to determine and return the plug-in which contains the OB associated with the node. If the node is not an operator, the function behaves as an oracle.

$$plugin(n) \equiv \begin{cases} \text{DetermineOperatorPlugin}(n), & \text{if } operator(n) \\ // \text{ Oracle}, & \text{otherwise.} \end{cases}$$

The result of this modification is that interpretation of any operator node encountered will be handled by the plug-in containing the appropriate OB. The DetermineOperatorPlugin rule performs the following procedure to determine which plug-in is suitable for interpretation of the given operator node:

1. Evaluate all operand subtrees of the operator node. Operand values are then immediately available for use.
2. Get the OD for this operator.
3. For each OB in the OD:
 - (a) Compute the result of its operation on the operand values.
 - (b) Store its result in a set. Along with the result, store a reference to the plug-in which provided the OB.

4. Decide if there is a valid result for this operator. The decision is based on examination of the set of results produced by the OD.
 - (a) If there is a valid result, return the plug-in containing the OB which provided the result. This plug-in will be given control of interpretation of the operator node.
 - (b) Otherwise, the engine has no plug-in equipped to handle the interpretation of the operator node.

This operator evaluation algorithm is described formally as follows

Interpreter: Operator Evaluation

```

DetermineOperatorPlugin( $n$  : NODE)  $\equiv$ 
  return pluginResponsible in
    obResults( $n$ ) := { }
    seq
      // Step 1
      EvaluateOperands( $n$ )
    seq
      // Step 2
      obSet( $n$ ) := getOD(token( $n$ ))
    seq
      // Step 3
      forall  $ob \in obSet$  do
        add ExecuteOB( $n, ob$ ) to obResults( $n$ )
    seq
      // Step 4
      pluginResponsible := DecideOnPlugin( $n$ )
  
```

where

```

getOD( $t$ )  $\equiv$  // Return set of all OBs for OD, given operator token
  
```

```

EvaluateOperands( $n$  : NODE)  $\equiv$  // Evaluate subtrees of operator pertaining to operands
  
```

```

ExecuteOB( $n$  : NODE,  $ob$  : OPERATORBEHAVIOR)  $\equiv$ 
  
```

```

  return (obValue( $n, ob$ ), obPlugin( $ob$ ))
  
```

where

```

obValue( $n, ob$ )  $\equiv$  // Return result of OBs computation on given node
  
```

```

obPlugin( $ob$ )  $\equiv$  // Return plug-in which provides OB
  
```

where EvaluateOperands is an abstract rule, and *getOD*, *obValue* and *obPlugin* are abstract functions.

The EvaluateOperands rule simply does what is necessary to evaluate the operand subtrees of the AST and then makes the operand values available for the OBs to use. The ExecuteOB rule executes an OB using the pre-computed operand values, and returns a 2-tuple consisting of the result of the operation and the plug-in to which the OB belongs.

Notice that all operand subtrees are evaluated before giving control to OBs. We considered giving a copy of each operand subtree to each OB and allowing the OB to have control of operand evaluation, but ruled out this approach as it is unnecessarily inefficient; Consider that for a single operator evaluation attempt, if an OD consists of k OBs, each operand subtree would be evaluated k times. Our approach evaluates each operand subtree only once regardless of the size of k . On first examination of this design decision, this seems to preclude the possibility of *lazy evaluation* of operands (the evaluation of operands only when needed). For example, during the evaluation of the Boolean conditional AND “&&” operator in Java, the second operand is evaluated only if the first evaluates to true; this operator is used in cases where, if the first operand is not true, an attempt to evaluate the second operand will result in an error. We feel that lazy evaluation of operands can be accommodated in CoreASM, but have only carried out a preliminary investigation of the issue. We have found that that the effects of lazy evaluation can be simulated in all cases except when the evaluation of an operand results in a non-terminating computation. To mitigate the case where an error results from evaluating an operator that in lazy evaluation would be evaluated only if necessary, error suppression can be used. In the case where a non-terminating computation occurs, it would be necessary to have a mechanism to detect and halt that computation. These issues are non-trivial, however, and so we leave them for future work.

To determine which plug-in should be responsible for the evaluation of the operator node, the collection of results must first be examined in order to determine whether the OD was sufficient for operator evaluation, and if there is a well-defined result for the operation. There is one case when operator evaluation is considered to be well-defined:

- All non-error results are equivalent. In this case we ignore all error results and assume that the OBs producing the errors are simply not equipped to handle operating on the given operands. However, what is important here is that all OBs that produce a

result, produce the same result.

The operator evaluation fails in all other cases. These cases can be summarized as follows:

- There are differing non-error results. This means that multiple OBs are equipped to handle the computation of a result but do not agree on the result. As such we cannot be sure of what the result of the operation should be.
- All OBs result in an error as a consequence of not being equipped to handle the given operands.

We note that a situation may arise where a failure in operator evaluation is caused by multiple OBs producing valid but conflicting results. Although the engine may not have the information it needs to choose the correct OB, the user might. In such cases, the ability to select a particular OB for the operator would be convenient. Different methods for selecting an OB have been discussed, and one proposed solution involves additional syntax allowing the user to specify the plug-in containing an OB for any particular use of an operator in a specification. For example, if there was an OB for “+” provided by the String Plug-in and another provided by the Number Plug-in, the user could force numerical addition by using the operator suffixed with the plug-in name (i.e. `+.Number`), and then only the OB from the indicated plug-in would be executed. We don’t formalize this here and leave the details of it for future work.

When operator evaluation is well-defined, any one of the plug-ins associated with an OB that produces a non-error result can be given control of the interpretation. The `DecideOn-Plugin` rule is the formal definition of the procedure used to determine the plug-in suitable for interpretation of an operator node:

Interpreter: Operator Evaluation

```

DecideOnPlugin( $n : \text{NODE}$ )  $\equiv$ 
  return  $pluginResponsible$ 
  if  $allSameResult$  then
    choose  $\langle v, p \rangle \in nonErrorResults$  do
       $pluginResponsible := p$ 
    else
       $pluginResponsible := undef$ 
  where
     $allSameResult \equiv \forall \langle v_1, p_1 \rangle \in nonErrorResults, \forall \langle v_2, p_2 \rangle \in nonErrorResults, v_1 = v_2$ 
     $nonErrorResults \equiv \{ \langle v, p \rangle \mid \langle v, p \rangle \in obResults(n) \wedge v \notin \text{ERROR} \}$ 

```

Notice that a failure of operator evaluation results in the engine being unable to determine which plug-in is responsible for the evaluation of this particular use of the operator.

6.2 Parser Extensibility

The Parser module of CoreASM handles both the lexical analysis and syntactic analysis stages of interpretation. Recall that the syntax of a language is specified using a grammar (see Section 3.2). Because computer languages generally do not change between major versions, their grammar is static. The CoreASM language is an oddball in this respect because it is required to be fully extensible by plug-ins with new rule forms, operators, and literals. As such, the grammar used to describe the language is dynamic. In this section we briefly describe how syntactic extensibility by plug-ins is facilitated.

6.2.1 The CoreASM Language Dependence on Specifications

In Section 4.1.2 we described the engine life-cycle. During the initialization stage of the engine, depicted in Figure 4.4, the `LoadStdPlugins` rule loads all standard plugins; these plugins contain grammar extensions. While in the process of loading a specification (depicted in Figure 4.5) in the *Parsing Header* mode of the engine, the header of the specification is parsed via the `ParseHeader` rule of the Parser. The `ParseHeader` rule looks for `use` directives which specify additional plug-ins to be loaded for use in execution of the specification (see Figure 6.1).

The engine mode then becomes *Loading Plug-ins*, where the loading of these additional

```
// Header
use Tree
use Map

// Body
...
```

Figure 6.1: An example use of **use** directives in a CoreASM specification. While the standard plug-ins are automatically loaded, the *Tree* and *Map* plug-ins are loaded especially for use with this specification.

plug-ins is done via the `LoadSpecPlugins` rule in the Control API. These additional plug-ins may also contain grammar extensions. Thus the CoreASM language syntax is dependent on the plug-in requirements of the specification to be interpreted, and hence can differ from one specification to another.

Once all plug-ins required by a specification have been loaded, the engine moves to the *Parsing Spec* mode of the engine, and executes the `ParseSpecification` rule:

```
ParseSpecification ≡
  BuildGrammar
  seq
  BuildAST
```

Parser: Parse Specification

At a high-level, the act of parsing the specification can be broken down into two major steps:

1. Building the grammar to use based on all grammar extensions provided by loaded plug-ins.
2. Using this grammar to building the AST which represents the specification.

Here we concentrate on how the first step is achieved. The second step involves typical lexical and syntactic analysis which we do not describe here. We direct the reader to [1] for more information on these stages of interpretation.

6.2.2 Dynamic Grammar

Recall that the kernel contains only the minimum functionality necessary for basic DASM semantics. The kernel provides a grammar which specifies the syntax for rules, operators, and literals to support these semantics. Regardless of which other plug-ins are required by a specification, the grammar included by the kernel is guaranteed to be present.

With this in mind, the *kernel grammar* has been structured in such a way that it is hierarchically segregated with respect to aspects of the language with extensible syntax.

```
Start -> ...  
...  
Rules -> ...  
Operators -> ...  
Literals -> ...
```

Figure 6.2: The kernel grammar structure showing GEPs.

In Figure 6.2 the general structure of the grammar is shown. The grammar is structured such that all productions describing rule form syntax are reached via the `Rules` production and all productions describing literal syntax can be reached via the `Literals` production. For each production extending the kernel grammar, we simply append that production to the RHS of the appropriate kernel grammar production. Productions which are designed to be extensible in this way are called *grammar extension points* (GEP).

For example, the Set Plug-In provides both new rule forms and literals. Assume that the productions which describe the Set Plug-In specific syntax are as follows:

```
...  
SetRules -> ...  
SetLiterals -> ...
```

Then the kernel grammar is extended at both the `Rule` and `Literal` GEPs with these additional productions:

```
Start -> ...
...
Rules -> ... | SetRules
Literals -> ... | SetLiterals
...
SetRules -> ...
SetLiterals -> ...
```

Extending the parser with new operators is a more complicated process that we describe here at a very high-level. In Section 3.4 we described how OP and operator associativity are encoded in the structure of the AST produced by a parse using a grammar. In CoreASM the segment of the grammar which describes operator syntax is built dynamically with operator classes, precedence levels and associativity in mind. This OPG is then integrated into the grammar via the `Operators GEP`. Upon loading plug-ins, the engine is made aware of the operators provided by each plug-in, as well as operator class (i.e. unary, binary, ternary, etc.), operator associativity (i.e. LA or RA) and operator precedence (which in the case of CoreASM is specified via a number between 0 and 100). Using all this information, the engine dynamically constructs the OPG productions required to properly describe the syntax and characteristics of all operators to be supported.

6.3 Summary

In this chapter we described CoreASM extensibility mechanisms for both operators and language syntax. With our black box approach to operator extension, an OB's operand requirements are hidden from the engine; we presented the procedure the engine uses to choose the correct OB without relying on sort information. We have also described how the structure of the kernel grammar is significant in facilitating language syntax extensibility, and have described how additional rule-form, literals, and operator syntax are accommodated.

“...and go on till you come to the end: then stop.”

— King of Hearts, Alice in Wonderland

Chapter 7

Conclusion and Accomplishments

We have presented in this thesis the design of several extensibility mechanisms and how they have been incorporated into the CoreASM engine and language for executable ASMs. Their presence will ensure that CoreASM, like the ASM formalism on which it is based, is flexible enough to accommodate application-specific data structures, operators and language syntax. Our specific accomplishments are:

- We have integrated into a CoreASM step the ability to aggregate incremental updates representing distributed partial modification of elements.
- Our aggregation framework is forward-compatible with data structure specific algorithms for aggregation and sequential composition, thus facilitating the introduction of new data types which require incremental update support.
- Our approach to operator extensibility allows for overloading without constraining the operator’s behaviours to specific data types. Our method of operator evaluation results in the correct OB being chosen without the engine being aware of the operands’ sorts.
- We have achieved syntactic flexibility through kernel grammar extension and dynamic grammar generation at run-time.

By adopting a black box approach to extending the engine with aggregation algorithms, composition algorithms, and operators, we have guaranteed that very few restrictions are imposed on such extensions, thus allowing CoreASM to meet the formalization needs of

nearly any problem domain. Using ASMs for specifying their formal requirements, we have given precise descriptions of both how these mechanisms function and the few requirements that must be met by plug-ins for the engine to accommodate their extensions.

The product of our efforts to integrate distributed incremental change of elements into the CoreASM engine is the first documented formal specification of both executable classic ASM and executable Turbo ASM steps with support for Gurevich's notion of partial updates. Moving from theory to practice has been no small feat, however the flexibility and freedom provided by the ASM formalism resulted in a precise specification for aggregation. This specification and the resulting implementation give further evidence that multi-agent incremental change can be achieved in ASMs without violating the original semantics of an ASM state transition.

By adhering to its tenets, our work has given further support to the feasibility of the CoreASM core ideology. The CoreASM engine itself forms the kernel of a novel environment for model-based engineering of abstract formal requirements and design specifications during the early stages of the software design and development process. The flexibility to extend this kernel will encourage and facilitate experimentation resulting in new tools to support the machine-aided exploration of many problem spaces.

7.1 Implementation and Project Involvement

The CoreASM project [27] is a Java-based Open Source software project which has grown into a collaborative effort with many active participants. Various design decisions for the engine and language have been influenced by our own input. Translating the true ASM semantics and syntax from the mathematical domain into a tangible tool while keeping it both pure and extensible was no trivial task. However, the modelling of CoreASM formal requirements using ASMs and the use of the specification as a guide have resulted in the speedy development of software which works as intended.

To date, the CoreASM engine kernel modules and engine extensibility mechanisms have been implemented. However the standard plug-in library planned for distribution with the kernel, and which uses the aforementioned extensibility mechanisms, is still under development.

We undertook many responsibilities in implementation. Using requirements described in the CoreASM specification, we designed and successfully implemented the entire Parser

module. We first determined the steps required to produce an AST from a specification while allowing for syntactic extensibility by plug-ins. Because the language syntax is dynamic, a parser must be generated for each specification based on the grammar produced by extending the kernel grammar at GEPs with productions provided by plug-ins. We designed an interface through which plug-ins are able to specify operator and grammar extension information, and implemented an algorithm to assemble all this information into the grammar used for parsing the specification.

After a broad search for parser generators, we decided upon the use of RIT OOPS [50], a novel parser generator that uses a fully object-oriented method to parse input [42]. Parser generators allow one to specify actions to be performed by the generated parser during the parsing process. These actions are triggered when nonterminals and terminals of the language grammar are recognized. In our case, each action contributes to the construction of an AST, the final result being a tree representing a CoreASM specification. While most parser generators require these actions to be specified inside the grammar source file, OOPS has facilities to encapsulate these actions in objects, thereby separating the grammar from the actions. This modularization of actions simplifies the syntactic extension process by allowing plug-ins to separately provide the Parser module both with grammar productions and with objects containing actions for building AST fragments for these productions as they are recognized.

In the Interpreter module, we implemented the operator evaluation framework and designed the OB interface that is used by plug-ins to extend the language with new operators. The operators provided by the Set Plug-In have been implemented and tested using that framework and interface.

We implemented the aggregation and incremental update framework in the Abstract Storage module and designed the aggregator and composer interfaces. The Set Plug-In, which introduces the background class of sets and uses the aggregation-related extensibility mechanisms formalized in this work, has been mostly¹ implemented as well.

Along with the accomplishments presented in this thesis, we have also time on the formalization and implementation of other standard plug-ins. These include both the Number Plug-in, providing the background class of decimal numbers, and the String Plug-in, providing the background class of character strings.

¹The implementations of set comprehension and the sequential composition algorithm for set incremental updates provided by the Set Plug-in are partially complete at the time of writing.

7.2 Future Work

Our framework for aggregation of distributed incremental change of elements has been incorporated into the engine. So far, this framework has been used by the set background. Further validation of the framework and additional guarantees of its stability can be achieved via both empirical testing and practical use, thereby uncovering possible inconsistencies or additional requirements that must be met. Directions for future work thus include the specification and implementation of additional background plug-ins that provide aggregatable sorts and use the aggregator and composer interfaces.

While the operator evaluation method introduced serves our immediate needs, it does not allow for all kinds of operators to be included in the engine. *CoreASM* is unable to support operators requiring lazy evaluation of operands since our operator evaluation method requires that all operands be evaluated before executing individual OBs. We believe that lazy evaluation of operands can be simulated in the engine by augmenting our approach, but this requires further exploration.

Finally, our operator evaluation method does not yet allow the user to explicitly request the use of a particular OB upon operator evaluation. Proposed solutions include prepending plug-in names with operator symbols, thereby requesting that the OB provided by the named plug-in be used for that particular use of the operator. Before adopting any approach the implications of such functionality must be studied.

Appendix A

CoreASM Component and Plug-In Interfaces

This appendix contains the interfaces provided by CoreASM components and plug-ins, which are not defined anywhere else in this document; these interfaces (i.e. ASM rules and functions) are used by the underlying machine. Note that only portions of each interface which are relevant to this thesis appear here. For more thorough coverage of CoreASM component and plug-in interfaces, the reader is directed to the most recent CoreASM technical report available¹.

A.1 Abstract Storage

The simulated state is modeled as a function $content : STATE \times LOC \rightarrow ELEMENT$, where locations are defined, as usual, by pairs of function names and arguments. All the functions in this section are **controlled** functions.

- $state : STATE$
is the current state of the simulated machine.
- $getValue : LOC \rightarrow ELEMENT$
returns the value of a given location. This function is defined as follows:

¹The latest CoreASM documentation can be downloaded from the project website [27]

$$getValue(l) = \begin{cases} content(state, l), & \text{if } content(state, l) \neq undef, \\ uu, & \text{otherwise.} \end{cases}$$

- **rule SetValue**($l : \text{LOC}, v : \text{ELEMENT}$)

sets the value of the given location to the given value. This rule is defined as follows:

$$\begin{aligned} \mathbf{SetValue}(l, v) &\equiv \\ &content(state, l) := v \end{aligned}$$

SetValue

- **rule PushState**

copies the current state in the stack. This rule is defined as follows²:

$$\begin{aligned} \mathbf{PushState} &\equiv \\ &asStack(asPtr) := state \\ &asPtr := asPtr + 1 \end{aligned}$$

Push

- **rule PopState**

retrieves the state from the top of the stack (thus discarding the current state). This routine is defined as follows:

$$\begin{aligned} \mathbf{PopState} &\equiv \\ &state := asStack(asPtr - 1) \\ &asPtr := asPtr - 1 \end{aligned}$$

Pop

- **rule Apply**($u : \text{UPDATESET}$)

applies the updates in the update set u to the current state.

$$\begin{aligned} \mathbf{Apply}(u) &\equiv \\ &\mathbf{forall} (l, v) \in u \mathbf{do} \\ &\quad \mathbf{SetValue}(l, v) \end{aligned}$$

Apply

- **isConsistent** : $\text{UPDATESET} \rightarrow \text{BOOLEAN}$

holds if the update set is consistent according to [15, Def. 2.4.5].

²We are assuming $asPtr = 0$ in the initial state.

A.1.1 Elements of the Superuniverse

The following functions are defined over all elements of the superuniverse.

- $bkg : \text{ELEMENT} \rightarrow \text{NAME}$
is the name of the background of the given element. The default value is "Element".
- $equal_{Element} : \text{ELEMENT} \times \text{ELEMENT} \rightarrow \text{BOOLEAN}$
returns *true* if the two elements are equal. We have

$$\forall a_1, a_2 \in \text{ELEMENT} \quad a_1 = a_2 \leftrightarrow equal_{Element}(a_1, a_2)$$

- **derived** $equal : \text{ELEMENT} \times \text{ELEMENT} \rightarrow \text{BOOLEAN}$
returns *true* if the given elements are equal. This function is defined as

$$equal(a_1, a_2) \equiv equal_{bkg(a_1)}(a_1, a_2) \vee equal_{bkg(a_2)}(a_2, a_1)$$

Notice that backgrounds which elements belong to provide sort specific methods for defining equality of their elements.

A.1.2 Element Enumerability

An enumerable element is any element which through some processing, can provide a collection of all the elements which constitute its internal structure. This general idea of enumerability can be easily applied to sets, a multisets, trees, records, etc.. The collection provided by these elements is a simple unordered group, which can contain duplicates.

The enumerable interface is useful as it provides a universal interface for all elements which can be represented (in albeit a simple form) by collection. The interface required by all enumerable elements is as follows:

- **controlled** $enumerable : \text{ELEMENT} \rightarrow \text{BOOLEAN}$
holds *true* if the element is enumerable, and *false* otherwise. The default value of this function is *false*.
- **derived** $enumerate : \text{ELEMENT} \rightarrow \text{ELEMENTCOLLECTION}$
provides a collection of elements contained within the internal structure of the enumerable element, and is defined by its background:

$$enumerate(e) \equiv enumerate_{bkg(e)}(e)$$

A.2 Set Plug-In

In evaluating the sets produced by set comprehension variant B, every possible combination of values for constrainer variables $x_{1..n}$ (given that the sets they are constrained to their constrainer expression $enumerate(value(\gamma_{1..n}))$ respectively) must be considered. Here we describe the rules used by these variants to generate all combinations.

The following function constitutes the simple structure used to keep record of considered combinations:

- **controlled** *consideredSet* : NODE \times INTEGER \times ELEMENT \rightarrow BOOLEAN
is used to keep track of possible values considered for a given constrainer variable x_i where $i \in [1..n]$.

The following rules describe the procedures used to do higher level tasks associated with book keeping, and combination choice:

- **rule InitializeConsideredCombos**
initializes *consideredSet* function in order to keep track of considered combinations of values for assignment to constrainer variables, chooses first combination to be considered, and assigns these ELEMENTS to their respective local variables.

InitializeChooseConsideredCombos

InitializeChooseConsideredCombos \equiv

```

forall  $i \in [1..n]$  do
  choose  $c \in enumerate(value(\gamma_i))$ 
  forall  $e \in enumerate(value(\gamma_i))$ 
    if equal( $e, c$ ) then
      consideredSet( $pos, i, c$ ) := true
      AddEnv( $x_i, c$ )
    else
      consideredSet( $pos, i, e$ ) := false

```

- **rule OtherCombosToConsider**
returns *true* if all combinations have not been considered at this point, and returns *false* otherwise.

OtherCombosToConsider \equiv

OtherCombosToConsider

```

return other in
  if  $\exists i \in [1..n]$   $\text{AllInSetConsidered}(i) = \text{false}$ 
    other := true
  else
    other := false

```

- rule ChooseNextCombo

chooses another combination to be considered, and assigns these ELEMENTS to their respective local variables.

ChooseNextCombo \equiv

ChooseNextCombo

```

ChooseNext(n)

```

- rule ClearConsideredCombos

resets data structures used for book keeping, and clear local variable definitions.

ClearConsideredCombos \equiv

ClearConsideredCombos

```

forall  $i \in [1..n]$  do
  forall  $e \in \text{enumerate}(\text{value}(\gamma_i))$ 
     $\text{consideredSet}(\text{pos}, i, e) := \text{false}$ 
  RemoveEnv( $x_i$ )

```

The following rules perform specific low level tasks, and are called by the higher level tasks described just above:

- rule AllInSetConsidered(i : INTEGER)

returns *true* if all elements in the set produced by the constrainer expression at γ_i have been considered for assignment to its respective constrainer variable x_i , and *false* otherwise.

AllInSetConsidered

```

AllInSetConsidered( $i : \text{INTEGER}$ )  $\equiv$ 
  return allConsidered in
    if  $\exists e \in \text{enumerate}(\text{value}(\gamma_i))$   $\text{consideredSet}(\text{pos}, i, e) = \text{false}$ 
      allConsidered := false
    else
      allConsidered := true

```

- **rule** **ChooseNext**($i : \text{INTEGER}$)

is a recursive rule which when called with $i = n$ causes the next unconsidered combination of elements to be chosen, and assigns these ELEMENTS to their respective local variables.

ChooseNext

```

ChooseNext( $i : \text{INTEGER}$ )  $\equiv$ 
  if AllInSetConsidered( $i$ ) = true then
    ResetChooseSetConsidered( $i$ )
    ChooseNext( $i - 1$ )
  else
    choose  $c$  in  $\text{enumerate}(\text{value}(\gamma_i))$  with  $\text{consideredSet}(\text{pos}, i, c) = \text{false}$  do
       $\text{consideredSet}(\text{pos}, i, c) := \text{true}$ 
       $\text{env}(x_i) := c$ 

```

- **rule** **ResetChooseSetConsidered**($i : \text{INTEGER}$)

resets the *consideredSet* function such that all elements of the set produced by the constrainer expression at γ_i are flagged to not have been considered, except one single element which will now be considered assigned to its respective constrainer variable x_i .

ResetChooseSetConsidered

ResetChooseSetConsidered($i : \text{INTEGER}$) \equiv
 choose $c \in \text{enumerate}(\text{value}(\gamma_i))$
 forall $e \in \text{enumerate}(\text{value}(\gamma_i))$
 if $\text{equal}(e, c)$ **then**
 $\text{consideredSet}(\text{pos}, i, c) := \text{true}$
 $\text{env}(x_i) := c$
 else
 $\text{consideredSet}(\text{pos}, i, e) := \text{false}$

Appendix B

How to Shoot Yourself in the Foot with CoreASM

“You write ‘shoot self in foot’, but it’s too abstract to execute, so you wind up having to define the gun, the bullets, the cylinder, your foot, your arm, your hand, the trigger, the firing pin, the ballistics, and your circulatory system, as well as how to load, aim and fire the gun. You then point the gun at your foot and pull the trigger. All six bullets in the cylinder are fired simultaneously, but the five not pointed down the barrel backfire, causing the gun to explode and blowing off your hand and lower arm. The sixth bullet shoots you in the foot.”

— Michael Letourneau

Bibliography

- [1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1988.
- [2] M. Anlauff. XASM – An Extensible, Component-Based Abstract State Machines Language. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines: Theory and Applications*, volume 1912 of *LNCS*, pages 69–90. Springer, 2000.
- [3] C. Beierle, E. Börger, I. Durdanovic, U. Glässer, and E. Riccobene. Refining Abstract Machine Specifications of the Steam Boiler Control to Well Documented Executable Code. In J.-R. Abrial, E. Börger, and H. Langmaack, editors, *Formal Methods for Industrial Applications. Specifying and Programming the Steam-Boiler Control*, volume 1165 of *LNCS*, pages 62–78. Springer, 1996.
- [4] A. Blass and Y. Gurevich. Background, Reserve, and Gandy Machines. In P. Clote and H. Schwichtenberg, editors, *Computer Science Logic (Proceedings of CSL 2000)*, volume 1862 of *LNCS*, pages 1–17. Springer, 2000.
- [5] Andreas Blass and Yuri Gurevich. Abstract State Machines Capture Parallel Algorithms. *ACM Transactions on Computation Logic*, 4(4):578–651, 2003.
- [6] E. Börger. A Logical Operational Semantics for Full Prolog. Part I: Selection Core and Control. In E. Börger, H. Kleine Büning, M. M. Richter, and W. Schönfeld, editors, *CSL'89. 3rd Workshop on Computer Science Logic*, volume 440 of *LNCS*, pages 36–64. Springer, 1990.
- [7] E. Börger. A Logical Operational Semantics of Full Prolog. Part II: Built-in Predicates for Database Manipulation. In B. Rován, editor, *Mathematical Foundations of Computer Science*, volume 452 of *LNCS*, pages 1–14. Springer, 1990.
- [8] E. Börger. Why Use Evolving Algebras for Hardware and Software Engineering? In M. Bartosek, J. Staudek, and J. Wiederman, editors, *Proceedings of SOFSEM'95, 22nd Seminar on Current Trends in Theory and Practice of Informatics*, volume 1012 of *LNCS*, pages 236–271. Springer, 1995.
- [9] E. Börger. The ASM ground model method as a foundation for requirements engineering. In *Verification: Theory and Practice*, pages 145–160, 2003.
- [10] E. Börger, G. Fruja, V. Gervasi, and R. F. Stärk. A high-level modular definition of the semantics of C#. *Theoretical Computer Science*, 2004. Accepted for publication.
- [11] E. Börger, U. Glässer, and W. Müller. The Semantics of Behavioral VHDL'93 Descriptions. In *EURO-DAC'94. European Design Automation Conference with EURO-VHDL'94*, pages 500–505, Los Alamitos, California, 1994. IEEE CS Press.

- [12] E. Börger, U. Glässer, and W. Müller. Formal Definition of an Abstract VHDL'93 Simulator by EA-Machines. In C. Delgado Kloos and P. T. Breuer, editors, *Formal Semantics for VHDL*, pages 107–139. Kluwer Academic Publishers, 1995.
- [13] E. Börger, P. Päppinghaus, and J. Schmid. Report on a Practical Application of ASMs in Software Design. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines: Theory and Applications*, volume 1912 of *LNCS*, pages 361–366. Springer, 2000.
- [14] E. Börger, E. Riccobene, and J. Schmid. Capturing Requirements by Abstract State Machines: The Light Control Case Study. *Journal of Universal Computer Science*, 6(7):597–620, 2000.
- [15] E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer, 2003.
- [16] Claus Brabrand and Michael I. Schwartzbach. Growing Languages with Metamorphic Syntax Macros. *ACM SIGPLAN Notices*, 37(3):31–40, 2002.
- [17] G. Del Castillo, Y. Gurevich, and K. Stroetmann. Typed Abstract State Machines. Unpublished manuscript, 1998.
- [18] G. Del Castillo. Towards Comprehensive Tool Support for Abstract State Machines. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Applied Formal Methods — FM-Trends 98*, volume 1641 of *LNCS*, pages 311–325. Springer, 1999.
- [19] G. Del Castillo and K. Winter. Model Checking Support for the ASM High-Level Language. In S. Graf and M. Schwartzbach, editors, *Proceedings of the 6th International Conference TACAS 2000*, volume 1785 of *LNCS*, pages 331–346. Springer, 2000.
- [20] E. Börger. High Level System Design and Analysis using Abstract State Machines. In D. Hutter and W. Stephan and P. Traverso and M. Ullmann, editor, *Current Trends in Applied Formal Methods (FM-Trends 98)*, volume 1641 of *LNCS*, pages 1–43. Springer, 1999.
- [21] E. Börger and W. Schulte. A Practical Method for Specification and Analysis of Exception Handling: A Java/JVM Case Study. *IEEE Transactions on Software Engineering*, 26(10):872–887, October 2000.
- [22] R. Eschbach, U. Glässer, R. Gotzhein, and A. Prinz. On the Formal Semantics of SDL-2000: A Compilation Approach Based on an Abstract SDL Machine. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines: Theory and Applications*, volume 1912 of *LNCS*, pages 242–265. Springer, 2000.
- [23] R. Farahbod, V. Gervasi, and U. Glässer. CoreASM: An Extensible ASM Execution Engine. *Fundamenta Informaticae XXI*, pages 1002–1039, 2005.
- [24] R. Farahbod, V. Gervasi, and U. Glässer. CoreASM: An Extensible ASM Execution Engine. In *Proceedings of the 12th International Workshop on Abstract State Machines*, 2005.
- [25] R. Farahbod, V. Gervasi, and U. Glässer. Design and Specification of the CoreASM Execution Engine. Technical Report SFU-CMPT-TR-2005-02, Simon Fraser University, February 2005.
- [26] R. Farahbod, V. Gervasi, U. Glässer, and M. Memon. Design and Specification of the CoreASM Execution Engine – Part 1 : The Kernel. Technical Report SFU-CMPT-TR-2006-09, Simon Fraser University, May 2006.
- [27] R. Farahbod, V. Gervasi, U. Glässer, M. Memon, et al. *The CoreASM Project*. <http://www.coreasm.org>.

- [28] R. Farahbod, U. Glässer, and M. Vajihollahi. Specification and Validation of the Business Process Execution Language for Web Services. In Wolf Zimmermann and Bernhard Thalheim, editors, *Abstract State Machines 2004. Advances In Theory And Practice: 11th International Workshop (ASM 2004)*, Germany, March 2004. Springer.
- [29] R. Farahbod, U. Glässer, and M. Vajihollahi. Abstract Operational Semantics of the Business Process Execution Language for Web Services. Technical Report SFU-CMPT-TR-2005-04, Simon Fraser University, Feb. 2005. Revised version of SFU-CMPT-TR-2004-03, April 2004.
- [30] R. Farahbod, U. Glässer, and M. Vajihollahi. A formal semantics for the business process execution language for Web Services. In Savitri Bevinakoppa, Luís Ferreira Pires, and Slimane Hammoudi, editors, *Web Services and Model-Driven Enterprise Information Systems*, pages 144–155, Portugal, May 2005. INSTICC Press.
- [31] Angelo Gargantini, Elvinia Riccobene, and Salvatore Rinzivillo. Using Spin to Generate Tests from ASM Specifications. In *Abstract State Machines 2003*, pages 263–277. Springer, 2003.
- [32] U. Glässer, R. Gotzhein, and A. Prinz. The Formal Semantics of SDL-2000: Status and Perspectives. *Computer Networks*, 42(3):343–358, June 2003.
- [33] U. Glässer and Q.-P. Gu. Formal Description and Analysis of a Distributed Location Service for Mobile Ad Hoc Networks. *Theoretical Computer Science*, 336:285–309, May 2005.
- [34] U. Glässer, Y. Gurevich, and M. Veanes. An Abstract Communication Architecture for Modeling Distributed Systems. *IEEE Transactions on Software Engineering*, 30(7):458–472, 2004.
- [35] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
- [36] Y. Gurevich. Sequential Abstract State Machines Capture Sequential Algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111, July 2000.
- [37] Y. Gurevich, W. Shulte, and M. Veanes. Towards Industrial Strength Abstract State Machines. Technical Report MSR-TR-2001-98, Microsoft Research, Oct 2001.
- [38] Y. Gurevich and N. Tillmann. Partial Updates: Exploration. *Journal of Universal Computer Science*, 7(11):917–951, 2001.
- [39] Y. Gurevich and N. Tillmann. Partial Updates. *Journal of Theoretical Computer Science*, 336(2-3):311–342, 2005.
- [40] Yuri Gurevich. May 1997 Draft of the ASM Guide. Technical Report CSE-TR-336-97, University of Michigan, 1997.
- [41] ITU-T Recommendation Z.100 Annex F (11/00). *SDL Formal Semantics Definition*. International Telecommunication Union, 2001.
- [42] Bernd Köhl and Axel-Tobias Schreiner. An object-oriented LL(1) parser generator. *ACM SIGPLAN Notices*, 35(12):33–40, December 2000.
- [43] Mark Lutz and David Ascher. *Learning Python, Second Edition*. O'Reilly Media, Inc., 2004.
- [44] Microsoft FSE Group. *The Abstract State Machine Language*. Last visited March 2006, <http://research.microsoft.com/fse/asml/>.
- [45] W. Müller, J. Ruf, and W. Rosenstiel. An ASM Based SystemC Simulation Semantics. In W. Müller, J. Ruf, and W. Rosenstiel, editors, *SystemC - Methodologies and Applications*. Kluwer, June 2003.

- [46] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.
- [47] R. Eschbach and U. Glässer and R. Gotzhein and M. von Löwis and A. Prinz. Formal Definition of SDL-2000: Compiling and Running SDL Specifications as ASM Models. *Journal of Universal Computer Science*, 7(11):1024–1049, 2001.
- [48] Joachim Schmid. *Executing ASM Specifications with AsmGofer*. Last visited March 2006, <http://www.tydo.de/AsmGofer/>.
- [49] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. William C. Brown Publishers, Dubuque, IA, USA, 1986.
- [50] A. Schreiner. *RIT OOPS*. Last visited March 2006, <http://www.cs.rit.edu/~ats/projects/oops/edu/doc/overview-summary.html>.
- [51] R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer, 2001.
- [52] Ming Su. Using Abstract State Machines To Model a Graphical User Interface System. Master's thesis, Simon Fraser University, Burnaby, Canada, Spring 2006.
- [53] Bruce A. Tate. *Beyond Java*. O'Reilly Media, Inc., 2005.
- [54] A. Zamulin. Typed Gurevich Machines Revisited. Joint CS & IIS Bulletin, Computer Science, 1997.
- [55] A. Zamulin. Object-oriented Abstract State Machines. In *Proceedings of the 28th Annual Conference of the German Society of Computer Science*. Technical Report, Magdeburg University, 1998.
- [56] A. Zamulin. Specification of Dynamic Systems by Typed Gurevich Machines. In Z. Bubnicki and A. Grzech, editors, *Proceedings of the 13th International Conference on System Science*, pages 160–167, Wroclaw, Poland, 15-18 September 1998.
- [57] A. Zamulin. Generic Facilities in Object-Oriented ASMs. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines – ASM 2000, International Workshop on Abstract State Machines, Monte Verita, Switzerland, Local Proceedings*, number 87 in TIK-Report, pages 426–446. Swiss Federal Institute of Technology (ETH) Zurich, March 2000.

Index

- abstract syntax tree, *see* AST
- add-to-rule**, 24
- aggregation, 62
 - incremental update, 62
 - basic update, 66
 - consistency, 71
 - phase, 65
 - plug-in, 68
 - consistency, 70
 - requirements, 71
 - responsibility, 68
 - regular update, 66
 - resultant update, 65
 - sequential composition, 73
 - plug-in responsibility, 74
 - update instruction, 63
 - update multiset, 65
- ASM
 - Basic, *see* Basic ASM
 - basic function, 8
 - basic relation, 8
 - control state, *see* control state ASM
 - distributed, *see* DASM
 - environment, 12
 - initial state, 8, 12
 - location, 8
 - main rule, 11
 - move, 12
 - operators, 96
 - program, 7
 - reserve, 9
 - rule, 8
 - run, 12
 - signature, 8
 - sort, 8, 95
 - state, 7
 - transition, 12
 - Turbo, *see* Turbo ASM
 - type conventions, 95–96
 - undef*, 8
 - universe, 8
 - update, 8, 9
 - update set, *see* update set
 - variable, 8
 - vocabulary, 8
- AST, 31
 - annotated, 40
 - evaluation, 31
 - traversal, 31
- background class, 21
- Basic ASM, 7–14
- block-rule**, 11
- choose-rule**, 10
- compiler, 28
- control state ASM, 19–21
- CoreASM
 - aggregation framework, 63–76
 - core ideology, 3
 - engine
 - commands, 43
 - components, 38, 40–42
 - execute specification, 42
 - kernel, 40, 50
 - mode, 43
 - plug-in, 40, 50–51
 - step, 39, 43–50
 - grammar
 - dynamic, 105–106

- extension point (*GEP*), 105
 - kernel, 105
 - language, 51
 - kernel, 56–57
 - notation, 51–56
 - pattern-action (*PA*) rule, *see* pattern-action (*PA*) rule
 - plug-in, 57–61
 - operator
 - behaviour (*OB*), 97
 - definition (*OD*), 97
 - evaluation, 98–103
 - extension, 96–98
 - plug-in
 - Set, 77–93
 - simulated machine, 37
 - type conventions, 96–98
 - underlying machine, 37
- DASM, 15–16
 - asynchronous, 16
 - agent, 15
 - program, 15
 - self*, 16
 - synchronous, 16
- denotational semantics, 25
- domain, 8
- environment, 12
- extend-rule**, 9
- forall-rule**, 11
- function
 - basic, 8, 12
 - controlled, 13
 - derived, 12
 - dynamic, 13
 - monitored, 13
 - out, 14
 - shared, 13
 - static, 13
- grammar, 29
 - nonterminal, 29
 - production, 29
 - start symbol, 29
 - terminal, 29
 - token, 29
- ifelse-rule**, 9
- import-rule**, 9
- interpreter, 28
- iterate-rule**, 15
- let-rule**, 18
- lexical analysis, 28
- local-clause**, 17
- location, 8
- named-rule**, 10
- operational semantics, 25
- operator, 32
 - associativity, 32
 - left (*LA*), 32
 - right (*RA*), 32
 - overloading, 34
 - precedence
 - grammar (*OPG*), 33
 - precedence (*OP*), 33
- par-rule**, 11
- parse tree, 30
- parsing, 30
- partial update, 22
 - integrate, 22
 - multiset, *see* partial update multiset
 - particle, 23
- partial update multiset, 23
 - consistent, 23
 - sequential composition, 24
- pattern-action (*PA*) rule, 53
 - symbols, *see* symbols
- remove-from-rule**, 24
- result-rule**, *see* rule \leftarrow
- return-clause**, 18
- rule, *see* ASM rule

- \leftarrow , 17
- add-to**, 24
- block**, 11
- choose**, 10
- extend**, 9
- forall**, 11
- ifelse**, 9
- import**, 9
- iterate**, 15
- let**, 18
- named**, 10
- remove-from**, 24
- seq**, 14
- skip**, 9
- update**, 8
- while**, 15
- semantic analysis, 28, 29, 34
- seq-rule**, 14
- skip-rule**, 9
- symbols
 - bkg*, 54
 - env*, 54
 - pos*, 52, 54
 - $[\cdot]$, 52
 - $[[pos]]$, 52, 54
 - α, β, \dots , 53
 - l*, 53
 - u*, 53
 - v*, 53
 - x*, 53
 - \square , 53
 - \sqcup , 53
 - \sqcap , 53
 - \boxplus , 52
 - \square , 53
- syntactic analysis, 28, 29
- token, 28, *see* grammar token
- Turbo ASM, 14–15
 - sequential composition, 14, 24
- type, 29, 34
 - casting, 35
 - coercion, 34
 - explicit conversion, 35
 - type checking, 29, 34
 - typing, 35–36
 - dynamic, 35
 - static, 35
 - strict, 35
 - strong, 35
 - weak, 35
 - universe, 8
 - update, 8, 9
 - update set, 12
 - apply, 12
 - consistent, 12
 - fire, 12
 - inconsistent, 12
 - sequential composition, 14
 - update-rule**, 8
 - where-clause**, 18
 - while-rule**, 15