Università di Pisa

Dipartimento di Informatica
Dottorato di Ricerca in Informatica

Ph.D. Thesis

On Modularity In Abstract State Machines

Simone Zenzaro

Supervisor
Prof. Vincenzo Gervasi

May 23, 2016

Abstract

In the field of model based formal methods we investigate the Abstract State Machine (ASM) modularity features. With the growing complexity of systems and the experience gained in more than thirty years of ASM method application a need for more manageable models emerged. We mainly investigate the notion of modules in ASMs as independent interacting components and the ability to identify portions of the machine state with the aim of improving the modelling process.

In this thesis we provide a language level semantically well defined solution for (1) the definition of ASM modules as independent services and their communication behaviour; (2) a new construct that operates on the global state of an ASM machine that ease the management of state partitions and their identification; (3) a novel transition rule for the management of computations providing different execution strategies and putting termination condition for the machine inside the specification; (4) a data definition convention along with a new transition rule for their manipulation via pattern matching.

In our work we build upon CoreASM, a well-known extensible modelling framework and tool environment for ASMs. The semantic of our modularity constructs is compatible with the one defined for the CoreASM interpreter. This ease the implementation of extension plugins for tool support of modularity features.

A real world system use case ground model ends the thesis exemplifying the practical usage of our modularity constructs.

To my family

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Managing the complexity of software systems

Software systems have become integral part of our daily life. We use computers for work and entertainment, televisions are now equipped with applications and internet access capabilities, smartphones mostly replaced the old mobile and are entirely based on collections of software applications, smart watches run Linux based operating systems, fit bands track our fitness activities and health related information. Even paper books are being substituted by their electronic counterparts. Trains are mainly controlled by software and the first fully software controlled cars are already available. Software has pervaded also the medical area where it has an important role in controlling surgical robots and even brain controlled limbs prostheses.

The diffusion of cloud based systems has also pushed the proliferation of web services that interacts with software applications. The increased development of distributed systems is further amplified by Internet of Things (IoT) and the Internet of Everything (IoE) where devices and applications cooperate to generate a highly interconnected system.

In this scenario system development is constantly growing in complexity. Ensuring that the software construction process results in a reliable and correct system is more than ever important. Also understanding the behaviour of distributed systems is becoming convoluted.

According to [65], software engineering is "a systematic approach to the analysis, design, assessment, implementation, test, maintenance and engineering of software, that is, the application of engineering to software". The field of software engineering is vast, we focus on some of the aspects of it that has been found useful to cope with system complexity and reliability. Probably the first well-known strategy to address difficult problems is *divide et impera* also called divide and conquer. This strategy has been applied in various forms from algorithms design and *Component Based Software Engineering (CBSE)* [53] to object oriented programming and modelling [78] or Aspect Oriented programming [61]. Abstracting from the specific applications of the divide et impera principle, the underneath idea is that system complexity can be managed by proper decomposition in smaller components, often referred to as *modules*. One of the consequences of good decomposition is the improvement of software reliability.

Reliability and dependability of a software system relates to the assessment of quality of the delivered system after its construction. According to [66] to achieve a dependable system, a set of methods for dependability procurement and validation are necessary. The dependability validation consists of error removal and error forecasting, so it depends on the evaluation of an already constructed system or a prototype for it. Fault tolerance [76] and fault avoidance instead focus more on the goal to "provide the system with the ability to deliver the specified service". Fault tolerance deals with the problem of how to provide resilience to the system when errors or unexpected behaviour occurs. Fault avoidance instead focuses on how to prevent *by construction* fault occurrences.

Formal methods are part of the fault avoidance strategies of software engineering. They repre-

sent a valid support to system analysis, understanding, specification and verification.

1.2 Subject Area

This thesis belongs to the Software engineering field. From its birth, software engineering is trying to apply engineering criteria to the process of building software. This goal is meant to provide the engineers the tools for creating good software and the users with trustworthy systems.

Software engineering research has focused on the goal to produce a design discipline for the the construction of software intensive systems. The need of such discipline comes from the request of reliable and dependable systems. A software system should provide the required functionalities and should be resilient.

Software systems are becoming larger and more elaborate over the years. The decomposition in modular, self contained components helps to manage such complexity and preserve the understandability of the system behaviour. A good design of the software permits to reduce the amount of code to write and provides as final result a solution to a problem. The first problem in software system construction is the requirement elicitation, in other words collecting and understanding the user requirements that define the future system. When the system has been developed, the problem shifts to the verification that what the system does corresponds to the intended behaviour and that the way it solves the initial problem is correct.

Formal methods are mathematically based techniques in software engineering that have been used to specify, develop and verify software and hardware systems. Their utility has been applied at different levels of the development process from the starting analysis to validation and also to reverse engineer legacy systems. There exists a large number of different formal methods. Most of them are designed to solve special problems like the automatic verification of system properties. Some provide graphical representations while others rely on mathematical formulas to describe a system.

In this thesis we focus on the Abstract State Machine (ASM) method: a model based formal method that emphasise freedom of abstraction. We analyse the method from the point of view of modular features. While there are examples of formal methods that provide modular features like object orientation or module definition, in the ASM language there are still pragmatic modularity issues.

1.3 Contribution

We propose modularity features in ASMs. We analyse the current definitions, techniques and solutions to provide system decomposition capabilities for the ASM method. A need for more manageable models emerged from the growing complexity of systems and from the experience gained in more than thirty years of ASM method application. Increasing complexity in specifications benefits from the ability to split the job of producing a model into smaller components possibly assigned to different groups of people. Currently the ASM method lacks proper support for modularity. We focus on the notion of modules in ASMs as independent interacting components and the ability to identify portions of the machine state with the goal to make models more manageable and closer to the problem domain improving the mapping between the real world and the specification describing it.

We provide a set of new constructs for:

- module definition as independent units of behaviour;
- state partitioning management;
- control flow modularization;
- data representation and decomposition by manipulation via pattern matching.

We provide the syntax and semantics for these constructs along with examples of usage. Our solutions propose a language level approach to enable modularity features from different perspectives. We build upon CoreASM [37], a well-known extensible modelling framework and tool environment for ASMs. In fact, we give the semantic of our modularity constructs as a calculus the extends the classic ASM language and as an interpreter that is compatible with the one defined for CoreASM. We have chosen to follow this path since CoreASM is one of the cornerstone of ASM tool support and provides a reliable mechanism for the extension of the language via plugins that will enable the support for our modularity features in the ASM framework. In order to exemplify the practical usage of our constructs, we provide a ground model leveraging modularity features for a European FP7 project in which we have contributed: the SeaClouds platform.

1.4 Plan of the thesis

The first part of this document introduces the background of our work. After we analyse the status of the ASM language identifying the motivation of our work. The central chapters discuss some modularity issues in the ASM language and propose solutions for them. In the final chapters we exemplify the usage of our contribution with a case study before concluding the document. The chapters content are organised as follows:

Chapter 2 We outline the context in which we will contribute, namely formal method application in software intensive analysis and development. We examine how formal methods can improve the software construction process. Then we focus on the ASM method. We recall the method core concepts of ground model and stepwise refinement. After that, we report the definition of ASM and its related concepts like basic ASMs, TurboASM, Control State ASMs, run of a machine, agents and distributed systems. A review of the current module support continues the chapter. The CoreASM framework is then presented with its extensibility capabilities and the description of its interpreter. Finally we introduce the notation for the semantic we will use throughout the rest of the thesis.

Chapter 3 In this chapter we begin with an historical overview of the ASM method from its origin to its current state outlining the evolution of the language. We then analyse some issues of the method that the long experience gained during the years of ASM method application have produced. Starting from these issues we describe our proposal for improving the language with modularity features.

Chapter 4 We investigate the status of control flow management in ASMs highlighting how its semantic has been reshaped multiple times in order to address changing needs in the modelling process. Then we introduce a construct for the modularization of computations along with some usage examples.

Chapter 5 We focus on the notion of module for ASM, a concept always needed in the modelling process and addressed in many different ways. We propose a service oriented construct for modules that will enable cooperative modelling and reuse of existing modules.

Chapter 6 This chapter deals with the drawbacks of global state management. We analyse how people coped with state isolation and we investigate different approaches to address this task. Then propose a construct to partition and manage the state.

Chapter 7 This chapter introduces the support for uniform representation of data values and their manipulation. A construct for data definition in the form of records and sum types is proposed. We highlight how the lack of proper support for such data have been a hurdle for some real attempts of modelling with ASMs. We present a definition for such kind of data that does not require parser extensions and introduce a pattern matching construct to manipulate them. We also propose a way to type check data in a typeless world.

Chapter 8 An example of usage of the novel modularity constructs that we propose for the ASM framework is provided. We describe the SeaClouds platform: a European FP7 project in the field of cloud computing. Then we model part of it using the new constructs producing a ground model for the system.

Chapter 9 In this chapter we summarise the contribution of the thesis an draw some conclusion and future work.

Chapter 2

Background

This chapter introduces the background concepts of the thesis. We begin with a quick overview on formal methods and then we focus on Abstract State Machines. We recall the definition for ASM method describing the ground model and the refinement concepts. After we give the formal definition of the ASMs describing basic, Control State, Turbo, and multi agent ASM definitions. Most of the definitions comes from the ASM Book [12]. The state of the art summary with regard to the notion of module in ASM is presented. The chapter concludes with the description of the CoreASM framework. These concepts will be given as a prerequisite for the rest of the document. The reader familiar with the ASM method can skip the first three sections. To understand completely the semantic definitions in the following chapters the notation details of the CoreASM interpreter should be read.

2.1 Formal Methods

According to Daniel M. Berry [8], a formal method is any attempt to use mathematics in the development of a software intensive computer-based system in order to improve the quality of the resulting system. So formal methods are mathematically based techniques for the specification, development and verification of software and hardware systems.

The role of formal method in the software engineering life-cycle is manifold because usually different methods focus on the formalisation of different aspects. So there are specific methods for the verification of system properties, for example properties on the expected behaviour. Another field of application of formal methods is requirement gathering, reasoning and coverage. Usually the formal method is used to create a formal specification that represent a precise model of the system. A *model* allows the system to be described in an abstract way and to reason about it[69]. The model become the starting point to discover requirements inconsistencies and find out implicit or missing assumptions. In fact requirement engineering [82] [74] [92] is one of the field in which formal methods have been recognised to be relevant. Depending on the formal language adopted, the model can describe the system in terms of precondition and post condition or as actual pseudo code that is also interesting for inspection. Reasoning about requirements in order to build a formal model of a system is also helpful for design purpose: the problem domain is understood better with the formal approach and the knowledge acquired influences positively the architectural decisions. Models represent also a form of precise documentation of the system and of all the design choices. Another interesting use of formal methods is in the reverse engineering of existing (legacy) systems for behaviour and architecture analysis and even impact analysis of system functionalities modifications.

For many years formal methods have been a debated topic since their application seemed an overkill for small systems[8]. The initial cost in terms of time and money spent to produce a model for the under subject system seemed exceeding the advantages of using the method. Moreover the lack of proper tool support had only produced formal models and verification for a subset of the

required features of systems. Initial models also required simplification in order to be automatically verified. So the effort of using formal method seemed simply not worth.

With the advancement in the state of the art for formal methods and the development of better supporting tools, formal method validity has been tested in many contexts [27] [16] with success and can be considered an indubitable added value in software engineering life-cycle.

2.1.1 Formal methods and languages overview

In literature a great variety of formal methods, languages and notations have been proposed to model systems. In this section we give a brief overview of them. Probably the most known modelling language is the Unified Modelling Language (UML). UML is commonly used in industrial software design and provides a visual language to represent models. However its semantic is not formally defined (although some attempts have been done in this direction [39]). For this reason it cannot be included among formal languages.

Formal language features are very diverse. Each formal language includes some of them and a clear separation between languages is not always possible. We are not going to give a complete classification of all the formal method features, but we just want to give the idea of the existing variety of formal languages by their characteristics.

The first type of classification we may attempt regards *model* versus *property* based methods. Property based methods include axiomatic and algebraic methods. Axiomatic methods are based on Hoare's work [55] that allow to specify a system in terms of its effects, the behaviour is specified indirectly through a set of axioms, preconditions and post-conditions that the specification must satisfy. An example of this kind of languages if Larch [52]. Algebraic methods are usually applied to describe concurrent and distributed systems. They may include temporal logic, like Language Of Temporal Ordering Specification (LOTOS) that is a formal specification language based on temporal ordering of events used for protocol specification of standards, or they can be based on *process calculi* or *process algebra*. Example of this formal languages are π-calculus[71] and *Communicating Sequential Processes* (CSP) [54]. Another language based on temporal logic is TLA+[64], it has been developed by Leslie Lamport to design, model, document and verify concurrent systems.

Model based approaches, instead, define a system constructing a model in terms of mathematical structures. This kind of models can be divided in *event based* or *state based* methods taking into account the representation of the model and its evolution. For state based methods, we can further split between simple and complex state approaches. In simple state approaches the state consists of simple predefined types for values, while complex state approaches allow arbitrary complex values in the state. Such values can be described by type constructors or by mathematical tools like Tarski structures. Among model based formal methods we recall the *Vienna Development Method* (VDM) [10] that is one of the older formal methods. It has been developed in the 70's and has been used in industrial applications. VDM focuses on the early stages of system development and its model provide the description of data and functionalities. The analysis of VDM model is mostly performed by inspection or mathematical proofs. Its extended version, VDM++[34], include also object oriented features. Z[83] is a notation based on the standard mathematical notation used in axiomatic set theory, lambda calculus, and first-order predicate logic. Inspired by Z, *Alloy*[58], provides a language for the specification of the behaviour of a system for which sets of constraints can be defined. Its tool support enable fully automatic analysis of software specifications.

Models described by mathematical formulas are also available. For example the Common Algebraic Specification Language (CASL)[9] is a general-purpose specification language based on first-order logic with induction.

Other model based languages are the approaches derived by the B method. The B method [2] and its evolution (Event-B[1]) are formal methods with tool-support based on an abstract machine notation. They are included in the simple state class, and are used for specification, design and automatic proof of system properties. A notion of refinement is also available to transforms models into actual code.

The Abstract State Machine method is also a model based formal approach to system specification. It is based on the notion of abstract machine with abstract state. It is a state based approach where the state can be arbitrary complex. The emphasis of this method is on the freedom of abstraction. It has been applied in industrial systems, programming language specification and many other fields. It provides the notion of refinement similarly to B. The verification of properties is performed manually by means of mathematical proofs of refinement steps or invariants.

All these languages provide a specification in terms of mathematical formulas or pseudo code. Other methods, instead, provide a graphical representation. For example the *Specification and Description Language* (SDL) that is a specification language targeted at the description of the behaviour of reactive systems, and *Petri Nets* for the description of distributed systems. Petri nets are included in the event based approaches.

Some formal methods allows to execute the specification for system simulation and testing, for example ASM models are in principle executable. In this thesis we will focus on the ASM method, mainly because we are interested in improving the modularity feature of the language. In the next section we introduce the ASM method more in detail.

2.2 The ASM method

The ASM method provides a good compromise between declarative, operational and functional views in modelling software systems and, in time, have been proved its relevance as in the software engineering process.

The ASM method "is a systems engineering technique which supports the integration of problem-domain-oriented modelling and analysis into the development cycle". The method consists of three concepts: the ASMs, the *ground model* and the *stepwise refinement*. In this section we focus on the last two concepts of the method while ASMs will be described in the following sections. The ASM method offers a uniform conceptual framework for requirement capture and system design. It can be applied both to software and hardware systems. Applying the method means to produce a ground model for the system that is a precise description of the system at the desired level of abstraction; then by stepwise refinement the model can be incrementally transformed into executable code. Notice that also the ground model (in general the ASMs) is in principle executable, the tools supporting ASMs usually offers a simulator that allow to run ASM models as prototype of the system itself prior to actual coding. The ground model provides the *contract* between the costumer and the software designer because it is intended to be precise enough to be useful to the designer yet understandable by non expert to allow correctness inspection and avoid conceptual and implementation mistakes.

2.2.1 The ground model

The *ground model* is an abstract state machine representing the *blueprint* of a system as an implementation independent, application oriented model. The ground model also represent a precise documentation of the system it describes and of the design decisions giving also an architectural overview useful for reasoning and understanding the system and for the analysis in case of changes and maintenance of its functionalities.

A ground model focuses on the solution of three main problems: the language and communication, the verification-method and the validation problems. The *language and communication* problem refers to the need to mediate between the application domain language and the mathematical world in which the model resides. The solution to this problem allows the generation of the software contract. The *verification-method* problem refers to the capability to verify if the model produced applying the method actually reflects the original intentions of the customer and is complete and consistent. The ability to inspect the model by the application domain expert ease the solution to this problem. Finally the *validation* problem refers to the capability for the model to be the basis on which a test plan for the system can be defined. In this sense module simulation for different relevant scenarios and model inspection provides the start point for the validation.

There are some properties that characterise ground models. A ground model is *precise*, *simple* in order to be inspected by domain experts, *concise* since compactness improves understanding the model, *abstract*, *complete* meaning that all the relevant features are present, *validatable* and rooted on precise *semantic foundation*.

2.2.2 Stepwise refinement

The idea of *stepwise refinement* comes from structured programming approach ([90] [33]). The *principle of substitutivity* is one of the established principles of refinement. It states that "it is acceptable to replace one program by another, provided it is impossible for a user of the programs to observe that the substitution has taken place". The ASM refinement method also integrates various more specific notions of refinement ([6], [5], [30], [32], [73], [72]). It is an instrument to assess the relation between two ASMs one of which is the more abstract and the other is the more specific (refined) one. With stepwise refinement an abstract model can be specialised down to the code that realises it (or the opposite). From another perspective, two different behaviour may be found to be one the refinement of the other allowing the replacement of one with the other. Also data can be refined to support different level of detail for the model. Each refinement step must be proved, this way they also represent a documented, step-by-step, validation of the implementation correctness. Figure 2.1 shows the ASM refinement scheme. Given two ASMs, M and M^*, and two states of M (S, S') and two states of M^* $(S^*, S^{*'})$, the relation $\equiv$ is an equivalence between sates[1]. The transition from one state to another in ASMs is performed via discrete steps. In the figure $\tau_1, \ldots, \tau_m$ and $\sigma_1, \ldots, \sigma_n$ are, respectively, the *computation segments* of M and M^*. Each computation segment corresponds to a step of the machine. The refinements is constituted by the relation between a subset of *"interesting"* states of the abstract machine with a subset of the refined machine states.

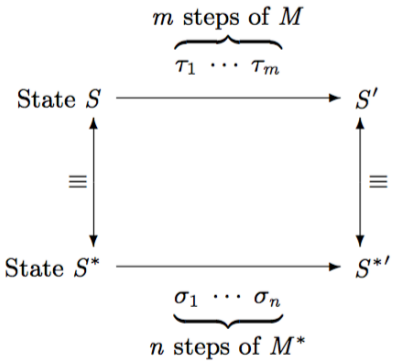


Figure 2.1: The ASM refinement scheme

More formally the definitions of correct and complete refinement are ([12]).

Definition 2.2.1. (Correct refinement) Fix any notion $\equiv$ of equivalence of states and of initial and final states. An ASM M^* is a correct refinement of an ASM M if and only if for each M^*-run $S_0^*, S_1^*, \ldots$ there is an M-run $S_0, S_1, \ldots$ and sequences $i_0 < i_1 < \ldots$, $j_0 < j_1 < \ldots$ such that $i_0 = j_0 = 0$ and $S_{i_k} \equiv S_{j_k}^*$ for each k and either

- both runs terminate and their final states are the last pair of equivalent states, or
- both runs and both sequences $i_0 < i_1 < \ldots$, $j_0 < j_1 < \ldots$ are infinite.

Definition 2.2.2. (Complete refinement) M^* is a complete refinement of M if and only if M is a correct refinement of M^*.

[1]For the definition of state see section 2.3.

2.3 ASM Definitions

2.3.1 Basic ASMs

Basic ASMs are tuples $\langle \Sigma, \mathcal{I}, \mathcal{R}, P_M \rangle$. The elements that constitute the tuple are:

Σ: The *signature* of the ASM. It is a finite set of function names f with an arity: a non negative integer, representing the number of arguments for the function. Functions with arity equal to zero are called *constants*. Every ASM signature includes the constants *true, false, undef*, where *undef* refers to the undefined value.

$\mathcal{I}$: A set of *initial states* for the signature. A *state* $\mathfrak{A}$ for the signature Σ is a non-empty set X, the *superuniverse* of $\mathfrak{A}$, together with *interpretations* of the function names of the signature. If f is an n-ary function name of Σ, then its interpretation $f^{\mathfrak{A}}$ is a function from X^n into X; if c is a constant its interpretation $c^{\mathfrak{A}}$ is an element of X.

$\mathcal{R}$: A set of *rule declarations*. A rule $\rho \in \mathcal{R}$ can be declared giving an expression in the form

$$r(x_1, \ldots, x_n) = P$$

where r is a rule name, P is a transition rule (see below) and its free variables are contained in the $x_1, \ldots, x_n$ list. Rule declarations can be evaluated to produce an *update set*. An update set is a set whose elements are of the form (l, v) where:

- l is a pair $(f, \langle a_1, \ldots, a_n \rangle)$ called *location* where f is a function name in Σ of arity n, and $a_1, \ldots, a_n \in |\mathfrak{A}|$ are elements of the superuniverse of the state.

- v is a value of the superuniverse $|\mathfrak{A}|$.

P_M: The *main rule* is an element of $\mathcal{R}$ distinguished from the others and with arity zero. It provides the *program* of the machine.

State transition

The behaviour of a system can be described by ASM models as the evolution of the machine state. The simulation of an ASM is the computation described by the, finite or infinite, sequence of consecutive state transitions

$$S_0 \xrightarrow{U_0} S_1 \xrightarrow{U_1} S_2 \xrightarrow{U_2} \ldots$$

.

The states of the machine (S_i), from the initial state S_0, are obtained evaluating the rules in the state and applying the relative update set U_i. The application of an update set consists of changing the value of all locations l in the update set U such that $(l, v) \in U$ to the new value v. The transition is allowed only for consistent updates. An update set is consistent if it has no clashing updates for any location. For each update in the update set U, if $(l, v) \in U$ and $(l, w) \in U$ then $v = w$. The transition from a state to the next is a *move* of the machine. A run of the machine is a sequence of moves.

For basic ASMs a set a basic transition rules is defined.

Skip: The **skip** rule semantic is to do nothing. It produces an empty update set.

Update rule: Writing $f(t_1, \ldots, t_n) := t$ means to assign the value of t interpreted in the current state to the n-ary function $f \in \Sigma$ at $a_1, \ldots, a_n$ where a_i are the values of t_i interpreted in the current state. The evaluation of an update rule is the singleton $\{(l, v)\}$.

Block Rule: The parallel execution of two transition rules is defined by the block rule as P **par** Q with the meaning of evaluating in parallel the transition rules P and Q. The evaluation of this rule results in the union of the update sets U_P, U_Q of P and Q.

Conditional Rule: The conditional rule **if** ϕ **then** P **else** Q first evaluates the guard ϕ. If it is true, the resulting update set is the evaluation of the P transition rule. Otherwise if is the result of evaluating Q.

Let Rule: **let** $x = t$ **in** P means to assign the value of t to the variable x and then to execute P. The evaluation results in the updates produced by P.

Forall Rule: Given a transition rule P, **forall** x **with** ϕ **do** P means to execute P for every x that satisfies ϕ. The resulting update set is the set produced by the parallel execution over x of P.

Choose Rule: **choose** x **with** ϕ **do** P **ifnone** Q means to non deterministically choose an x that satisfies ϕ and for that x execute P. If there is no x that satisfies ϕ execute Q. The resulting update set is the update set of the executed rule (P or Q).

2.3.2 Notation for the ASM semantic

The denotation $[\![^{\alpha}P]\!]^S_\zeta$ for a rule P is the update set that results from the interpretation of P in the state S with the variable assignment ζ. If the variable assignment ζ does not change we simplify the notation writing $[\![P]\!]^S$. The notation α denotes the rule identifier. Rule identifiers permit to make a distinction between rules with the same syntax that appears in different places inside an ASM model. In order to get a rule identifier we assume that there exists an oracle Ω that, given a rule P, returns the instance $\Omega(P) = \alpha$ as a location name. In tool environments, such kind of oracle may be easily encoded by a function that returns the node identifier from, for example, the abstract syntax tree representation for the ASM specification.

The same notation can be used to denote the interpretation of terms and formulas. The interpretation of a term t is defined by cases as:

1. $[\![x]\!]^S_\zeta = \zeta(x)$ for the variable x;
2. $[\![c]\!]^S_\zeta = c^S$ for the constant c;
3. $[\![f(t_1, \ldots, t_n]\!]^S_\zeta = f^S([\![t_1]\!]^S_\zeta, \ldots, [\![t_n]\!]^S_\zeta)$ for functions.

Formulas of a machine signature Σ are generated as follows. The semantics of formulas is defined by the table 2.1.

1. If s and t are terms of Σ, the $s == t$ is a formula;
2. If ϕ is a fomula, then the its negation $\neg\phi$ is a formula;
3. If ϕ and ψ, then their conjunction $(\phi \wedge \psi)$ is a formula;
4. If ϕ and ψ, then their disjunction $(\phi \vee \psi)$ is a formula;
5. If ϕ and ψ, then the implication $(\phi \rightarrow \psi)$ is a formula;
6. If ϕ is a formula and x a variable, then $(\forall x\phi)$ a $(\exists x\phi)$ are formulas.

Given two update sets U and V, the notation $U \oplus V$ denotes the merge of two update sets and is defined as:

$$U \oplus V = \begin{cases} \{(loc, val) \in U | loc \notin Locs(V)\} \cup V & \text{if } U \text{ is consistent;} \\ U & \text{otherwise} \end{cases}$$

Where *Locs* is the function that returns the set of locations of an update set. The notation $S + U$ denotes a state obtained from S applying the updates of the set U. It is defined for each location l as:

$$(S + U)(l) = \begin{cases} v & \text{if } (l, v) \in U \\ S(l) & \text{otherwise} \end{cases}$$

where $S(l)$ denotes the value of l in the state S.

$$
\begin{array}{lll}
[\![s = t]\!]^S_\zeta & = \begin{cases} true, & \text{if } [\![s]\!]^S_\zeta = [\![t]\!]^S_\zeta \\ false, & \text{otherwise} \end{cases} \\
[\![\neg\phi]\!]^S_\zeta & = \begin{cases} true, & \text{if } [\![\phi]\!]^S_\zeta = false \\ false, & \text{otherwise} \end{cases} \\
[\![\phi \wedge \psi]\!]^S_\zeta & = \begin{cases} true, & \text{if } [\![\phi]\!]^S_\zeta = true \text{ and } [\![\psi]\!]^S_\zeta = true \\ false, & \text{otherwise} \end{cases} \\
[\![\phi \vee \psi]\!]^S_\zeta & = \begin{cases} true, & \text{if } [\![\phi]\!]^S_\zeta = true \text{ or } [\![\psi]\!]^S_\zeta = true \\ false, & \text{otherwise} \end{cases} \\
[\![\phi \rightarrow \psi]\!]^S_\zeta & = \begin{cases} true, & \text{if } [\![\phi]\!]^S_\zeta = false \text{ or } [\![\psi]\!]^S_\zeta = true \\ false, & \text{otherwise} \end{cases} \\
[\![\forall x\phi]\!]^S_\zeta & = \begin{cases} true, & \text{if } [\![\phi]\!]^S_{\zeta[x\rightarrow a]} = true \text{ for every element } a \text{ of the superuniverse of } S \\ false, & \text{otherwise} \end{cases} \\
[\![\exists x\phi]\!]^S_\zeta & = \begin{cases} true, & \text{if there exists an } a \text{ in the superuniverse of } S \text{ with } [\![\phi]\!]^S_{\zeta[x\rightarrow a]} = true \\ false, & \text{otherwise} \end{cases}
\end{array}
$$

Table 2.1: The semantics of formulas

2.3.3 Control State ASMs

Control state ASMs are a particular class of abstract state machines. It can be considered a generalisation of finite state automata.

A control state ASM is an ASM whose rules are all of the form

if $ctl\_state = i$ **then**
  if $cond^i_1$ **then** R^i_1
  if $cond^i_2$ **then** R^i_2
  ...
  if $cond^i_n$ **then** R^i_n

The location $ctl\_state \in 1, \ldots, m$ is the control state of the machine that identifies the *"internal states"* of the machine, while the conditions $cond^i_k$ control which rule (R^i_k) must be executed. All the conditions are evaluated in parallel and, if no condition is satisfied, the machine does nothing[2]. Figure 2.2 shows the graphical notation for control state ASMs.

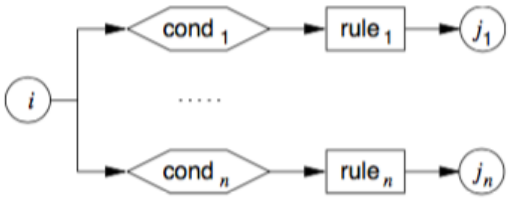
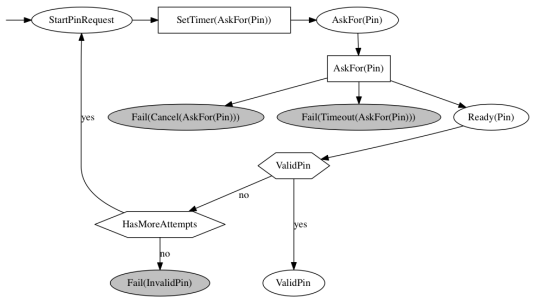


Figure 2.2: Control state ASM graphical notation

Such a notation allows to draw ASM diagrams that actually are flow diagrams representing the behaviour of the ASM. These diagrams usually are more understandable by non experts than their pseudo-code version. An example of control state ASM diagram is shown in Figure 2.3 that shows a portion of the control state ASM constituting the ground model for an automated teller

[2]More precisely, if the *else* branch of the conditional expression is not present it is assumed to be equal so **else skip**

machine (ATM) in the phase of processing the correctness of the pin inserted see [24] for the full model. Rounded shapes correspond to values of the control state, blocks are transition rules and diamond shapes are conditions.



Employee example

if $mode = StartPinRequest$ **then**
  SetTimeout(AskFor(Pin))
  $mode := AskFor(Pin)$
if $mode = AskFor(Pin)$ **then**
  AskFor(Pin)
if $mode = Ready(Pin)$ **then**
  if $ValidPin$ **then**
    if $HasMoreAttempts$ **then**
      $mode = StartPinRequest$
    else
      $mode := Fail(InvalidPin)$
  else
    $mode := ValidPin$

Figure 2.3: Process pin control state ASM

Notice that control state ASM are "only" a convenient organisation of basic ASMs. The imposed structure is helpful to isolate subsets of the system behaviour by its internal states. Composing each condition with the first level *if* (the one on the ctl state) generates a basic ASM that is equivalent to the original one at the cost of worse readability.

2.3.4 Turbo ASMs

Turbo ASM are an extension of basic ASMs that permit the composition of machines. They introduce three operators for: *sequential* composition, *iteration* and *parameterized submachines*. Turbo ASMs are a black-box view of the combined machine that hides the the detail of its internal behaviour. From an external point of view (of the combined machine) the execution of a Turbo ASM occurs in *one* single step although, internally, the Turbo ASM may have executed several steps. Since the computation of a machine may diverge (it requires an infinite number of steps to be completed), composing a divergent Turbo ASM will result in an undefined behaviour.

Definition 2.3.1. (Turbo ASM) Any ASM obtained from basic ASMs by applying finitely often and in any order the operators of sequential composition, iteration and submachine call is a Turbo ASM.

Sequential composition

The sequential composition of two machines P and Q is denoted by P **seq** Q. It is defined as

$$[\![P \textbf{ seq } Q]\!]^S_\zeta = [\![P]\!]^S_\zeta \oplus [\![Q]\!]^{S+[\![P]\!]^S_\zeta}_\zeta$$

So if the evaluation of P produces a consistent update set, the machine Q is evaluated in the state in which $[\![P]\!]^S_\zeta$ is applied and the resulting update set provides the final result.

Iteration

Starting from the sequential operator, if it is applied to the same machine repeatedly it is possible to define an iteration operator. From the nature of the sequential operator, the iteration can be stopped only in two cases: when the produced update set is inconsistent or empty. So the operator for iteration is defined by.

Definition 2.3.2. (Iterate Operator) $[\![\textbf{iterate } R]\!]^S = \lim_{n\to\infty}[\![R^n]\!]$ if for some $n \geq 0$ holds that $[\![R]\!]^{S_n}$ is empty or inconsistent. Where S_n is the state obtained by applying the update set produced by R_n in state S and R^n is recursively defined as:

$$R^n = \begin{cases} \textbf{skip} & \text{if } n = 0 \\ R^{n-1} \textbf{ seq } R & \text{otherwise} \end{cases}$$

Submachines

ASM submachines have been introduced to allow recursive calls to machines. They are defined by a notation extension of the macro technique providing parameterized machines. The definition of submachine is given only when the chain of nested calls is finite. The definition of a submachine is given declaring a parametric rule $R(x_1, \ldots, x_n) = body$. Where $x_1, \ldots, x_n$ are the formal parameters of the submachine and *body* is the rule with free variables the parameters that define the submachine behaviour. R is the name of the submachine.

The transition rules are extended with rule calls. A rule call is denoted by $R(a_1, \ldots, a_n)$ where R is the name of a declared parametric submachine and $a_1, \ldots, a_n$ are the actual parameters passed by name replacing the formal parameters.

Definition 2.3.3. (Turbo submachine) Let R be a named rule declared by $R(x_1, \ldots, x_n) = body$ and let S be a state. If $[\![body[a_1/x_1, \ldots, a_n/x_n]]\!]^S$ is defined, then

$$[\![R(a_1, \ldots, a_n)]\!]^S = [\![body[a_1/x_1, \ldots, a_n/x_n]]\!]^S$$

Local state

Turbo submachines also introduce the concept of local functions. At declaration time, a parametric machine may contain a set of local function definitions with a corresponding initialisation rule. On rule call, the function initialisers are called before the actual body evaluation. The rule call declaration becomes:

$$R(x_1, \ldots, x_n) = \textbf{local } f_1[Init_1], \ldots, \textbf{local } f_k[Init_k]body$$

So the rule R has its own definition for the functions $f_1, \ldots, f_k$. The restriction of the scope is achieved discarding, at the end of the rule call, all the updates concerning the local functions.

Definition 2.3.4. (Turbo ASM with local functions) Let R be a rule declaration with local functions as given above. The two terms in the following equation are either both undefined or both defined and equal:

$$[\![R(a_1, \ldots, a_n)]\!]^S = [\![(\{Init_1, \ldots, Init_k\} \textbf{ seq } body)[a_1/x_1, \ldots, a_n/x_n]]\!]^S \setminus Updates(f_1, \ldots, f_k)$$

Where *Updates* is returns the set of updates generated for its parameters.

Error handling

From the observation that programming languages usually offer some kind of error handling constructs that allow to separate error handling procedures from the normal execution of code, Turbo ASMs introduce a form of error handling support. In the ASM world an exception, abstractly, is represented by an inconsistent update. Given two turbo ASMs, P and Q, and set of terms T, writing **try** P **catch** TQ means to execute P and if the produced update set is not consistent on the location $t \in T$ execute Q. More formally:

Definition 2.3.5. Let P and Q be turbo ASMs and T a set of terms,

$$[\![\textbf{try } P \textbf{ catch } TQ]\!]^S = \begin{cases} [\![P]\!]^S & \text{if it is consistent on } Locs(T) \\ [\![Q]\!]^S & \text{otherwise} \end{cases}$$

Return values

Turbo ASMs with return values have been introduced to allow rule calls to behave as function calls. Submachine parameters are considered the input to the function and a special global dynamic function is responsible for storing the function output value. The reserved zero-ary function **result** is introduced to store the output value computed inside a submachine. The notation $l \leftarrow R(a_1, \ldots, a_n)$ denotes that the execution of the R submachine produce an update for the location **result** whose value is assigned to the location l.

Definition 2.3.6. (Machine with return value) Let $R(x_1, \ldots, x_n) = body$ be a rule declaration.

$$[\![l \leftarrow R(a_1, \ldots, a_n)]\!]^S = [\![body[l/\textbf{result}\ , a_1/x_1, \ldots, a_n/x_n]]\!]$$

2.3.5 Multi-agent ASMs

All the ASMs described in this chapter till now belong to the single agent ASM class. The ASM method is also meant to support another modularization mechanism for the design of large systems. Multi-agent ASMs provide a way to describe a system from a component view to analyse their interaction, moreover they are useful to describe the behaviour of distributed systems. There are two kind of multi-agent ASMs: synchronous and asynchronous.

Before describing the two versions of multi-agent ASM we want to recall what an agent is in the ASM method.

The notion of Agent has been introduced in the Lipari guide [45], and is an element of a dynamic set *Agents*. Elements of the *Agents* set are couples $\langle a, \pi \rangle$ of agent names (a) with the associated *program* (π). A program for an agent is a basic or a turbo ASM.

Synchronous multi-agent ASMs

A multi-agent synchronous ASM is a set of agents which execute their own program in parallel, synchronised using an implicit global system clock. The behaviour of a synchronous multi-agent ASM is equivalent to an agent whose program is the parallel execution of all the programs of the agents concurring to the definition of the multi-agent ASM. The agents may have different signatures, but the communication between agents occurs modifying the value of common locations that constitute the communication interface.

Asynchronous multi-agent ASMs

Asynchronous multi-agent ASMs are a set of Agents whose program run in the local state identified by the reserved *self* location. Each function and rule of the agent program signature is parameterized with *self*. For a function f the expression $f(self)$ indicates the "private" version[3] of f that belongs to agent self.

Since all the agents composing an asynchronous multi-agent ASM run their own program independently with different clocks, step moments and duration, a different definition of run that relates single agents executions is needed.

The run of an asynchronous ASM is a *partially ordered run*, that is an ordered set $(M, <)$ of rule applications m (moves) of its agents satisfying three conditions:

- *finite history*: for all $m \in M$ the set $\{m' | m' < m\}$ is finite. In other words each move has finitely many predecessors,

- *sequentiality of agents*: exists a function $<$ that linearize the set of agents moves $\{m | m \in M, a \in Agents$ performs $m\}$,

- *coherence*: each final segment X of $(M, <)$ has an associated state $\sigma(X)$, that corresponds to the result of applying in order $(<)$ all the moves in X, which for every maximal element $m \in X$ is the result of applying m in the state $\sigma(X - \{m\})$.

[3] Actually the global function with the addition of *self* parameter as first argument, *self* value is usually the identity of the agent

2.3.6 ASM modules

Syntactical declaration of modules

The following definition introduces a mechanism to syntactically structure large ASMs. The module definition in the ASM book [12] consists of a pair: the *Header* and the *Body*. The body of an ASM module contains function and rule declarations for the module. A module header consists of the name of the module, its import and export clauses, and its signature:

asm module

MODULE m
IMPORT $m_1(id_{1l}, \ldots, id_{1l_1}), \ldots, m_k(id_{k1}, \ldots, id_{kl_k})$
EXPORT $id_1, \ldots, id_e$
SIGNATURE s

The names $id_{1l}, \ldots, id_{1l_1}$ are functions or rules imported from another module m_i, and $id_1, \ldots, id_e$ are the names for functions or rules which can be exported from module m. The signature s of a module contains all the basic functions occurring in the module and all the functions which appear in the parameters of any of the imported modules. This definition assumes that there are no name clashes in these signatures. Notice that this assumption makes the definition impractical: during the modelling process each module should be free to choose the names that better suits the module domain.

Every ASM M becomes an ASM module if its main rule is added to the declarations and its name appears in the export list.

Asmundo composition operators

Restricting to basic ASMs, in [4], Asmundo and Riccobene address the problem of composing specifications. They define two operators for machines composition with the goal to provide a mechanism to safely combine specifications developed separately. With regards to safety they consider the absence of conflicts with regard to the machine signatures. The first operator is meant to be used for *feature composition* and is denoted by the symbol $\oplus$, while the second is denoted by $\otimes$ and refers to *component composition*. Both operators are applied to two sequential ASMs but the first is meant to describe a device as the composition of its features while the second combine components of a system.

The $\oplus$ operator is defined only for sequential ASMs that meet compatibility, consistency and safe tests. In the following definitions Υ_i is the signature of the machine, Π_i is a program. The notation $F_i(M)$ stands for the set of functions of M in the class i, where the possible classes are Out, Mon for monitored, $Ctrl$ for controlled, Sh for shared functions. If the safe test fails, an unsafe test must be checked.

Definition 2.3.7 (Compatible sequential ASMs)**.** Let $M_1, \ldots, M_n$ be sequential ASMs. They are *compatible* if for every $f \in \Upsilon_i \cap \Upsilon_j$, $i, j \in \{1, \ldots, n\}, i \neq j$, if dynamic, f belongs to the same function class in M_i and in M_j.

If the ASM machines to combine are compatible they can be combined without update inconsistencies only if a safe test succeeds. The safe test is defined as:

Definition 2.3.8 (Safe Test)**.** For every couple of indexes $i, j \in \{1, \ldots, n\}$, $i \neq j$ and for every

$$f \in (F_{Out}(M_i) \cup F_{Ctrl}(M_i) \cup F_{Sh}(M_i)) \cap (F_{Out}(M_j) \cup F_{Ctrl}(M_j) \cup F_{Sh}(M_j))$$

do the following:

- let $R_i^1, \ldots, R_i^l$ be all the rules in M_i updating f
- let $R_j^1, \ldots, R_j^m$ be all the rules in M_j updating f
- let $g_i^h, h \in \{1, \ldots, l\}$, and $g_j^k, h \in \{1, \ldots, l\}$, $k \in \{1, \ldots, m\}$, be the guards of the rules above.

If $\varphi = \bigvee_{h,k}(g_i^h \wedge g_j^k)$ is a ground contradictory formula, then there are no simultaneous updates for f.

Definition 2.3.9 (Unsafe test)**.** Check whether the composition M is a model for φ or not.

If $M \nvDash \varphi$ then there are not simultaneous updates of the same function so the composition is consistent with respect to updates. The first operator is then defined as:

Definition 2.3.10 ($\oplus$-Composition for Features of a Same Device)**.** Given n compatible sequential ASMs $M_1, \ldots, M_n$ with vocabularies $\Upsilon_1, \ldots, \Upsilon_n$, the $\oplus$-composition of $M_1, \ldots, M_n$ is the sequential ASM $M = \oplus_{i=1}^n M_i$ constructed as follows:

1. the vocabulary of M is $\Upsilon = \bigcup_{i=1}^n \Upsilon_i$ and the classes of dynamic functions of M are constructed by the function classes of the components in this way: $F_{Class}(M) = \bigcup_{i=1}^n F_{Class}(M_i)$, for $Class = Mon, Ctrl, Out, Sh$;

2. the program of M is Π: **do in parallel** $\Pi_1, \ldots, \Pi_n$, where $\Pi_1, \ldots, \Pi_n$ are respectively the programs of $M_1, \ldots, M_n$;

3. the set of states, $\mathcal{S}(M)$, is a set of Υ-structures X constructed from a (not necessarily unique) n-uple $X_1, \ldots, X_n$ of states of $M_1, \ldots, M_n$ such that:

   - there exists an $i \in \{1, \ldots\}$ such that X_i is the reduct of X to Υ_i,
   - and for every $f \in \Upsilon \setminus \Upsilon_i$, f is interpreted in X as X_j for some j such that $f \in \Upsilon_j$;

   $\mathcal{S}(M)$ is closed under isomorphism;

4. the set of initial states $\mathcal{I}(M) \subseteq \mathcal{S}(M)$ is constructed from $\mathcal{I}(M)$ (as in step 3.); $\mathcal{I}(M)$is closed under isomorphism;

5. the one-step transformation of M is the map τ_M to $\mathcal{S}(M)$.

The second operator is defined in two versions, one for the composition of synchronous components, the other version instead is for asynchronous systems.

Before introducing the composition operation for synchronous components, we present the condition that must be meet when the operator $\otimes$ is applied to machines where the self function is replaced by the name of the agent running their program. A notion of compatibility is given also for this second operator.

Definition 2.3.11 (Compatibility)**.** $M_1', \ldots, M_n'$ are *compatible* if for every $f \in \Upsilon_i' \cap \Upsilon_j'$, $i \neq j$, f has the same arity in Υ_i' and in Υ_j'.

The composition of machines can be tested with respect to updates applying the following function test.

Definition 2.3.12 (Function Test)**.** If the following condition are satisfied, then the composition of $M_1', \ldots, M_n'$ is consistent with respect to updates:

- for every $f \in \bigcup_{i=1}^n \Upsilon_i'$ if $f \in F_{Ctrl}(M_i')$, $i \in \{1, \ldots, n\}$, then $f \notin (F_{Ctrl}(M_j') \cup F_{Sh}(M_j') \cup F_{Mon}(M_j') \cup F_{Out}(M_j'))$ for every $j \in \{1, \ldots, n\}$, $i \neq j$;
- $(F_{Out}(M_i') \cup F_{Sh}(M_i')) \cap (F_{Out}(M_j') \cup F_{Sh}(M_j')) = \varnothing$ for every $i, j \in \{1, \ldots, n\}$, $i \neq j$.

If this test fails, it is possible to apply the safe and unsafe test defined previously on the machine result of the combination $\otimes$ that is defined as:

Definition 2.3.13 ($\otimes$-Composition of System Components Moving Synchronously)**.** Given n compatible synchronised sequential ASMs $M_1', \ldots, M_n'$, the $\otimes$-composition of $M_1', \ldots, M_n'$ is the multi-agent ASM $M = \otimes_{i=1}^n M_i'$ with synchronised agents $a_1, \ldots, a_n$, constructed as follows:

1. a finite indexed set of programs $\Pi'_1, \ldots, \Pi'_n$(modules) named as $\nu_1, \ldots, \nu_n$;

2. the vocabulary $\Upsilon = \bigcup_{i=1}^{n} \Upsilon'_i\{Mod, \nu_1, \ldots, \nu_n\}$, the function Mod assigns to each agent its program name; dynamic function symbols in Υ are classified as follows:

   (a) $F_{Sh}(M) = \bigcup_{i,j=1}^{n,i\neq j}(\{f | f \in F_{Mon}(M'_i) \cap F_{Out}(M'_j)\} \bigcup \bigcup_{i=1}^{n} F_{Sh}(M'_i)$

   (b) $F_{Mon}(M) = \bigcup_{i=1}^{n} F_{Mon}(M'_i) \setminus F_{Sh}(M)$

   (c) $F_{Out}(M) = \bigcup_{i=1}^{n} F_{Out}(M'_i) \setminus F_{Sh}(M)$

   (d) $F_{Ctrl}(M) = \bigcup_{i=1}^{n} F_{Ctrl}(M'_i)$

3. a collection of Υ-structures being the *states* of M which are constructed from the $\mathcal{S}(M')$ in this way: for every n-uple $X'_1, \ldots, X'_n$ of states of $M'_1, \ldots, M'_n$ such that all the shared locations are identically evaluated, construct a Υ-structure X which interprets symbols in Υ as $X'_1, \ldots, X'_n$.

4. a collection of Υ-structures being the *initial state* of M constructed from $\mathcal{I}(M_i)$ as in step 3.

The asynchronous version of the same operator is given by:

Definition 2.3.14 (Consistency)**.** A multi-agent ASM M with asynchronous agents is *consistent* with respect to updates if it cannot happen that from a state X of M, moves x and y from agents a and b are independently fired and x and y respectively originate updates $(f, \bar{a}, b_2)$ and $(f, \bar{a}, b_2)$ such that $b_1 \neq b_2$ with $f \in \Upsilon$ vocabulary of M, a, b_1, b_2 elements of X.

Definition 2.3.15 ($\otimes$-Composition of System Components Moving Asynchronously)**.** Given n compatible not synchronised sequential ASMs $M'_1, \ldots, M'_n$, the $\otimes$-composition of $M'_1, \ldots, M'_n$ is the multi-agent ASM $M = \otimes_{i=1}^{n} M'_i$ with asynchronous agents $a_1, \ldots, a_n$ constructed as follows:

1. a finite indexed set of programs $\Pi'_1, \ldots, \Pi'_n$ (modules) named $\nu_1, \ldots, \nu_n$;

2. the vocabulary $\Upsilon = \bigcup_{i=1}^{n} \Upsilon'_i \bigcup\{Mod, \nu_1, \ldots, \nu_n\}$, Mod assigns to each agent its program name; Υ has the same dynamic function classification as the synchronous case;

3. a collection of Υ-structures being the states of M which are constructed from the $\mathcal{S}(M'_i)$ as in the synchronous case;

4. a collection of Υ-structures being the initial states of M constructed from $\mathcal{I}(M_i)$.

Contract based modular refinement for submachines

In [36] Ernst et al. have defined another notion of module[4] for a variant of the ASMs. Their work has been motivated by their effort to construct a verified file system for flash memory. Inspired by contract refinement in Z [91] they adapted the notion of *contract refinement* to ASMs.

The working definition of ASMs for their work are *data type-like* ASMs.

Definition 2.3.16. A (data type-like) ASMs $M = (SIG, Ax, Init, \{Op_j\}_{j\in J})$ consists of a signature SIG, a set Ax of predicate logic axioms for the static part of the signature, a predicate $Init$ to characterise initial states, and a set of operations for indices $j \in J$. Each operation $Op_j = (pre_j, \underline{in}_j, \alpha_j, \underline{out}_j)$ consists of an ASM rule α_j that describes possible state transitions, provided precondition pre_j holds. It reads input from a vector $\underline{in}_j$ of input locations, and writes output to a vector $\underline{out}_j$ of output locations. It may modify local variables, controlled locations and the locations of $\underline{out}_j$. The rules should have no non-local variables.

[4] To be more precise of modular refinement

It is interesting to notice that there are different point in which is required to have care in crafting the operation rules. The reasons are justifiable but from a practical point of view, using data-like ASMs requires a supplementary effort for the specifier.

The definition of run for such machines is given from an atomic and non atomic perspective. Here we recall the definition of atomic semantic for the operation that allow the definition of a run for data type-like ASMs.

Definition 2.3.17. The atomic semantics $[\![OP]\!] \subseteq S_\perp \times S_\perp$ of an ASM operation $Op = (pre, \underline{in}, \alpha, \underline{out})$ is defined as:

$(s, s') \in [\![Op]\!]$ if and only if either $s \neq \perp$ and $s \not\models pre$ (and s' is arbitrary from $S_\perp$) or $s \neq \perp, s \vDash pre$ and there is I with $I(0) = s, I \vDash \alpha$ and if I is finite then $s' = I.last$, otherwise $s' = \perp$ or $s = s' = \perp$.

Where the semantic of rules is assumed to be executed non-atomically. The notation $I = (I(0), I(1), \dots)$ introduces finite or infinite sequences called intervals, with the meaning that if $I \vDash \alpha$, I is a possible execution of α.

A run of a machine is so defined by:

Definition 2.3.18. An ASM program over a machine M is a possibly infinite sequence $\underline{j} = (j_0, j_1, \dots)$ of (indices or names of) operation calls. An execution of the program $\underline{j}$ is an interval I with states in $S_\perp$ and $\#I = \#\underline{j}$, where

$(s, s') \in [\![Op_{j_k}]\!]$ for $s = I(k)$ and $s' = I(k+1)\{\underline{in}_{j_k} \mapsto s(\underline{in}_{j_k})\}$

holds for all $0 \leq k < \#I$. An execution is a run of the program, written $I \in runs^M(\underline{j})$ if it starts with an initial state $I(0) \neq \perp, I(0) \vDash Init$.

In this definition the interval I may contain $\perp$, and $\#I$ stands for the cardinality of the interval. The notion of submachine allow from a machine M to call the operation of another submachine. Given two machines $M = (SIG, Ax, Init, \{Op_j\}_{j \in J})$ and $L = (SIG^L, Ax^L, Init^L, \{Op^L_k\}_{k \in K})$, M can call the operation of L if:

- M extends L's signature and axioms: $SIG^L \subseteq SIG$ and $Ax^L \subseteq Ax$,
- initialisation of M includes initialisation of L
- M respects information hiding: the signature of L is never accessed directly by operations of M (M can only access SIG^L via operation calls)

The operation call is performed passing the input parameters for the operation by value and by copying the outputs to the specified caller locations. The call of an operation is considered as one atomic step.

The notion of refinement of such machines is given in order to be able to replace an abstract machine with a concrete one that has the same operations of the abstract machine. It is called *contract* refinement for ASMs.

The approach presented in this work is really promising, although the definition of the machine require to be really careful about locations access and introduces yet another variant of ASMs.

2.4 CoreASM

CoreASM [37] is an extensible modelling framework and tool environment that support the ASM method. It is focused on the early phases of the software design process providing the instruments to rapidly building system prototypes with ASMs. Models are built with the CoreASM language, an untyped abstract language that is very close to the standard ASM language. The focus on prototyping allows to experiment with the system design producing abstract models that can be later refined to more concrete ones. Although abstract, CoreASM models are mathematically-precise and executable in order to allow the simulation of prototypes.

One peculiar characteristic of the language is extensibility: a plugin-based mechanism allows the language to be extended with new constructs. This feature has been emphasised a lot since in the history of ASM, as we have discussed in chapter 2, new transition rules and syntactic sugar enriched the language.

CoreASM is an open source project continuously updated and maintained by part of the ASM community.

The framework principal component is the *engine* that provide an interpreter of CoreASM specification. In the following sections we describe the architecture of CoreASM, how the interpreter executes the specification and introduces the notation to specify the semantic of new plugin rules. This last information is at the basis of the semantic definitions of our modularity constructs.

2.4.1 Engine

The CoreASM engine is the heart of the framework. It consists of four macro components: a parser, an interpreter, a scheduler and an abstract storage (see Figure 2.4). The cooperation of these components allows CoreASM to execute ASM models. The component *Control API* provide the interface to instruct the engine and control the simulation, for example it is possible to load a specification, start a run, execute a single step of the simulated machine, pausing the simulation and so on. All the tools available for the CoreASM environment are bound to the Control API. There exists a plugin for the Eclipse IDE, a debugger, a command line interface to the engine, graphical user interfaces and other applications that together constitute the CoreASM framework.

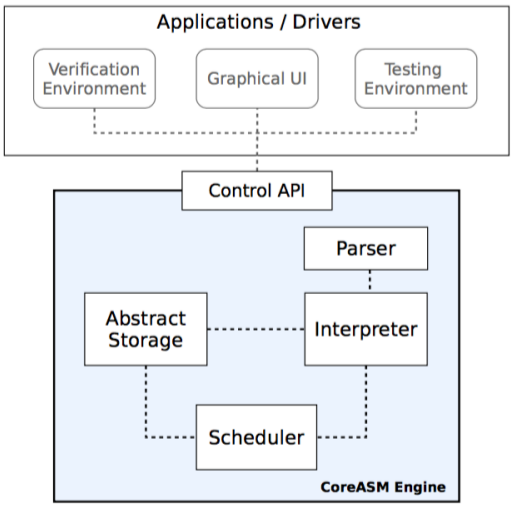


Figure 2.4: CoreASM architectural view

Each component of the engine has its own specific function in the simulation of ASM models. The parser reads a model specification and generate from it an annotated abstract syntax tree for the rules and definitions of the specification (see Figure 2.5). The tree is the input of the *interpreter* that communicates with the *abstract storage* in order to get the values from the current state and evaluates the rules to generate update sets. The *abstract storage* manages the state of the simulated machine maintaining also the history of the simulation keeping the previous states. For each step of execution, the *scheduler* orchestrate the evolution of the computation choosing the agents of the simulated model accordingly to a specified policy.

The simulation of a model is a complex process that requires various steps. The first step is the engine initialisation that prepares it to execute every CoreASM specification. Then a specification must be loaded into the engine that parses it and retrieve all the necessary components (plugins) that are needed by the specification. The abstract storage is initialised setting up the initial state. At this point, through the control API, *step* commands start the simulation. The execution of a move for the simulated machine starts with the scheduler that selects a (subsets) of agent that will concur to the computation of the next step. For each of the selected agents, the interpreter proceed with the evaluation of their programs that corresponds to traversing the parse tree of the

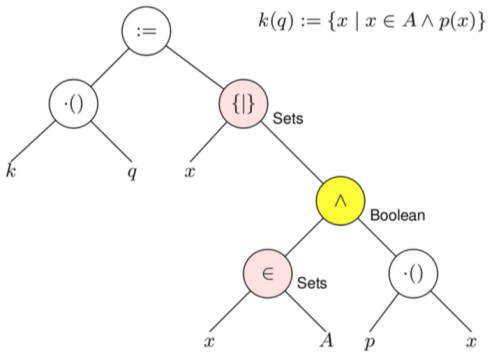


Figure 2.5: Parsing tree

program. The result is a set of update instructions that, if found consistent, updates the state of the machine. These steps are reiterated until a termination condition is met. With regard to termination conditions, CoreASM permits to specify them outside the specification itself as a property of the execution.

Plugins

The architecture of CoreASM is also identified by the *kernel* of the engine and the set of extension plugins. Figure 2.6 shows this view of the CoreASM architecture. The *kernel* represents the minimal set of functionalities that are needed to execute the most basic ASM models. Kernel functionalities can be extended providing plugins. CoreASM supports three classes of plugins: *backgrounds*, *rules* and *policies*.

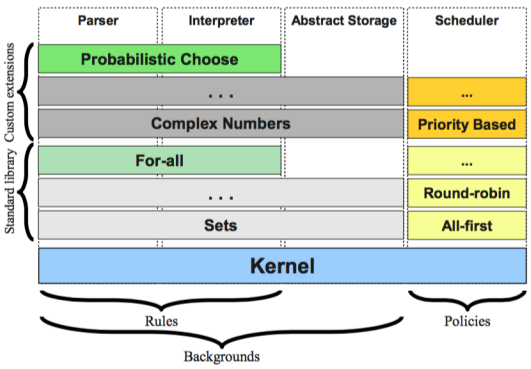


Figure 2.6: CoreASM architecture layers

Background plugins provide extensibility of the domains the language can use in the specification. Example of domains are sets, lists, maps, numbers. A plugin in this class specify the nature of the elements belonging to the background, the concrete syntax to operate on them, the corresponding semantic and some functions for encoding and decoding that will be used by the abstract storage to manage these elements inside the simulated machine state.

Rule plugins extend the set of constructs that generates updates. The syntax and semantic for new rules must be provided as an extension of the parser and of the interpreter. For example, in CoreASM also the *if* statement is not inside the kernel but it is implemented as a rule plugin for the engine.

Policy plugins extend the *scheduler* strategies for the selection of the agents that, at each step, move forward the machine computation. CoreASM allow only a single policy plugin to be chosen for a computation. Changing the policy require to modify the specification and to choose another policy.

The CoreASM engine provides an already defined *standard library* of plugins that implements the most common background plugins (e.g. numbers, sets, maps, strings, bags) and all the transition rules for the ASMs described in [12].

2.4.2 The interpreter

The CoreASM language is specified by its interpreter. As we already mentioned, the plugin architecture allows to extend the language adding new transition rules and backgrounds. When a specification is loaded into the engine, all the plugin needed to parse and interpret it are loaded as well. The interpreter is constituted by the collection of ASM rules from kernel and plugins. After the parser generates the abstract syntax tree representation of the specification, each engine step triggers a traverse of it. The tree nodes are evaluated and update sets are collected during the process. The nodes contain a reference to the plugin in charge of its interpretation. In order to navigate the tree the function *first, next, parent* has been defined. The function $first$: NODE $\rightarrow$ NODE points to the first child of a node, while *next* : NODE $\rightarrow$ NODE accesses to the next sibling of the parameter node. To access the parent node the interpreter applies the function *parent* : NODE $\rightarrow$ NODE. The variable *pos* holds the current evaluation position of the interpreter in the abstract syntax tree. Another class of functions are available to retrieve information about nodes. The function *class* : NODE $\rightarrow$ CLASS returns the syntactical class of a node. Examples of classes for a node are literals, identifiers and rule declarations. The syntactical token of the node is returned by the function *token* : NODE $\rightarrow$ TOKEN. Each grammar rule has associated a symbolic name that is accessible via the function *pattern* : NODE $\rightarrow$ PATTERN. Information about the plugin responsible for parsing and evaluating the node is referred by the *plugin* : NODE $\rightarrow$ PLUGIN.

The evaluation function is denoted by $[\![.]\!]$: NODE $\rightarrow$ LOC $\times$ UPDATES $\times$ ELEMENT. The result of evaluating a node is a triple consisting of a location, a multiset of update instructions and a value. To access the single elements in the triple, the three projection function *loc, updates, value* are provided. Testing the evaluation status of a node is possible applying the function *evaluated* : NODE $\rightarrow$ BOOLEAN defined as $evaluated(n) \equiv [\![n]\!] \neq undef$.

Accessing the value of local variables to get their value is performed by the environment function *env* : TOKEN $\rightarrow$ ELEMENT The association for each value to its background is maintained by the function *bkg* : ELEMENT $\rightarrow$ BACKGROUND.

Semantic rules for the interpreter usually require to define complex conditions on the abstract syntax tree nodes. CoreASM defines a concise form of pattern matching to denote such conditions. A rule pattern is written in the form

$$(\!|\, pattern \,|\!) \rightarrow \quad actions$$

and is to be intended as

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

Where the *conditions* are derived from the pattern according to the of table 2.3. In the *action* part of the rule, occurrences of unquoted and unbound occurrences of l, u, v and x are to be interpreted as, respectively, the *loc*, *updates*, *value* and *token* of the corresponding node.

The following conventions define the condition on nodes for rule patterns:

- denotes a generic node;
- denotes a generic unevaluated node, the semantically equivalent versions of it , and are defined as an aid for the reader and their meaning denotes that the expected result of the evaluation is a value from an expression, a set of updates from a rule or a location;

- x denotes an identifier node;
- v denotes an evaluated expression node whose *value* is not $undef$;
- u denotes an evaluated statement node whose *updates* are not $undef$;
- l denotes an evaluated expression for which a location has been computed (*loc* is not $undef$);
- Greek letters denote positions in the parse tree (typically children of the *pos* current node), for example in **if** $^{\alpha}$[?] **then** $^{\beta}$[?] the letters α and β denotes the first condition node and the then block of an if statement.

As an example we now show how a typical definition of the semantic for a rule to extend the ASM parser, we show how to define with the pattern rule conventions the semantic of the if-then statement. In Table 2.2, the left side contains the rule definition while the right side shows the correspondent expansion of the conditions on the nodes.

Compact notation	Actual rule
$(\!\|$ **if** $^{\alpha}$[e] **then** $^{\beta}$[r] $\|\!)$ $\rightarrow$ $pos := \alpha$	**if** $class(pos) \neq \mathsf{Id}$ $\wedge\, pattern(pos) = \mathsf{IfThen}$ $\wedge\, class(first(pos)) \neq \mathsf{Id}$ $\wedge\, \neg evaluated(first(pos))$ $\wedge\, class(next(first(pos))) \neq \mathsf{Id}$ $\wedge\, \neg evaluated(next(first(pos)))$ **then** $pos := alpha$

$(\!\|$ **if** $^{\alpha}v$ **then** $^{\beta}$[r] $\|\!)$ $\rightarrow$ **if** $^{\alpha}v = true$ **then**
$pos := \beta$
else
$[\![pos]\!] := (\mathtt{undef}, \{\!\|\!\}, \mathtt{undef})$
	if $class(pos) \neq \mathsf{Id}$
	$\wedge\, pattern(pos) = \mathsf{IfThen}$
	$\wedge\, value(first(pos)) \neq \mathtt{undef}$
	$\wedge\, class(next(first(pos))) \neq \mathsf{Id}$
	$\wedge\, \neg evaluated(next(first(pos)))$
	then
	if $value(first(pos)) = true$ **then**
	$pos := next(first(pos))$
	else
	$[\![pos]\!] := (\mathtt{undef}, \{\!\|\!\}, \mathtt{undef})$

$(\!\|$ **if** $^{\alpha}v$ **then** $^{\beta}u$ $\|\!)$ $\rightarrow$ $[\![pos]\!] := (\mathtt{undef}, {}^{\alpha}u, \mathtt{undef})$	**if** $class(pos) \neq \mathsf{Id}$ $\wedge\, pattern(pos) = \mathsf{IfThen}$ $\wedge\, value(first(pos)) \neq \mathtt{undef}$ $\wedge\, updates(next(first(pos))) \neq \mathtt{undef}$ **then** $[\![pos]\!] := (\mathtt{undef}, updates(next(first(pos))), \mathtt{undef})$

Table 2.2: Rule pattern for if statement semantic

Throughout the rest of this thesis we will use the same style of semantic definition for the constructs we introduce as modularity features. We could use any other formalism but this approach has the advantage of building up on the top of the solid CoreASM semantic and prepare the definitions to be integrated in the framework in the form of plugins. This way the support for our constructs becomes complete.

Abbreviation	**Condition**	**Mention**
α, β, etc.		*first*(pos), *next(first(pos))*, etc.
$^{\alpha}\boxed{?}$	$class(\alpha) \neq \mathsf{Id}$	
$^{\alpha}\boxed{\ }$	$class(\alpha) \neq \mathsf{Id} \wedge \neg evaluated(\alpha)$	
$^{\alpha}\boxed{\in}$,$^{\alpha}\boxed{\Gamma}$, $^{\alpha}\boxed{1}$	$class(\alpha) \neq \mathsf{Id} \wedge \neg evaluated(\alpha)$	
$^{\alpha}x$	$class(\alpha) = \mathsf{Id}$	$token(\alpha)$
$^{\alpha}v$	$value(\alpha) \neq undef$	$value(\alpha)$
$^{\alpha}u$	$updates(\alpha) \neq undef$	$updates(\alpha)$
$^{\alpha}l$	$loc(\alpha) \neq undef$	$loc(\alpha)$

Table 2.3: Abbreviations in syntactic pattern-matching rules.

Chapter 3

Proposal

In this section we recall the history of ASMs to understand why we think the ASM method should be improved, then we analyse what the experience that applying the method have brought to light highlighting some drawbacks of the method. The chapter concludes with a proposal of improvement for the ASM method that will be further explored in the next chapters. Our goal is to boost modularity features of the language in order to make specifications more manageable when modelling complex software intensive system.

3.1 The history of ASMs

The ASM method has a long history, from the very first discovery of the ASM concept to its adoption as a valuable formal method in software engineering the ASMs have gone through different phases.

The ASM concept have been first discovered by Yuri Gurevich [43][47] in an attempt to improve on the *Church-Turing thesis*. At that time Gurevich used the term **dynamic structures**. The main goal was to include resource boundaries considerations for the Chruch-Turing thesis. Later on the name *evolving algebras* took the place of dynamic structures. The first real test for the ASM as a formal method came in 1988 when a complete and precise yet compact semantic for Prolog has been defined in terms of ASMs [13, 14, 15]. The Prolog full semantic definition is one of the first significant example of ASM specification that paved the way to the recognition of practical relevance of the method. In that case also the concept of *stepwise refinement* have been introduced. The idea of stepwise refinement allows the model description to be defined at different levels of abstraction, for the Prolog model stepwise refinement provided a mathematically provable relation between the high level view of the model down to the software implementation details.

Although the Prolog model has to be considered an important contribution to the method assessment, looking now at the model produced for Prolog makes wonder how structured it was: the model has been presented as a set of flat **if - then** statements with the only structure being the textual paragraph in which they were presented. In fact the notion of module at that time overlapped with the label for a set of **if** statements and a textual description to relate them.

The experience with Prolog led to further experiment ASM potentialities in documenting design decisions and capturing requirements giving birth to the *ground model* notion, in other words a model at the abstraction level required by the problem domain itself. The nature of *pseudo code over abstract data* of ASMs also started to incorporate the idea of executable models into the method. An executable model can simulate a system prior to coding in an attempt to reduce the errors during its development.

In 1991 Gurevich gave the first definition of *sequential ASMs* [46] that will be completed in the Lipari guide [45]. Such definition can be considered the foundation for the proof of the ASM thesis [48].

We recall that a sequential ASM is a set of guarded rules in the form

if g **then** R

here g is a boolean guard and R is a set of update instructions. An update instruction is an expression in the form $f(\bar{t}) := t_0$ with f being a function, $\bar{t}$ a tuple of terms and t_0 another term. In the Lipari guide definition the core elements of ASMs where already present. In particular the concept of update, the parallelism of updates as default behaviour, the notion of *run*, and Tarski structures to represent abstract state. In such definition also the difference between *internal* and *external* functions has appeared. The external functions are function that only the environment may change. In this case the environment is seen as everything outside the machine, while the internal functions represent the machine *global* state.

Distributed ASMs are also part of the Lipari guide definition bringing the notion of agent as a *module* identified by a name and a program. The definition of agent already shows in embryo the need for a *local state* relative to the agent.It refers to two special functions $self$ and $View$. The $self$ function is an attempt to define the identification of the state relative to an agent. The $View$ function is exactly defined as "local state of agent a corresponding to the global state".

In those years, a great variety of ASM models for programming language definitions have been produced, notably a semantic for C [49], COBOL[88], Smalltalk[11].

The flourishing number of successful programming language specification was the lever to further test the practicability of ASMs in different contexts [23, 89, 22, 56, 50]. A good survey of these works can be found in [16].

The ASMs where tested against architecture design problems such as the control unit processor *zCPU*[18, 25]. Also virtual machines have been specified, an interesting example is the Parallel Virtual Machine (PVM) for which a ground model has been defined in [20, 21]. Other problems in which the ASM has been found to be a successful approach are compilation process correctness (see [77]). Testing the ASM method has also been achieved through protocols modelling, for example in [22] three ASMs describe in a readable but precise way the Lamport's mutual exclusion protocol [62, 63].

During the history of the ASM method a set of tools for their support has been developed. Tool support is an important part of a formal method since they ease the adoption of the method and reduce the cost of application of its. Among them we recall ASMGofer [79] an extension of the functional programming language Gofer [80] with parallel update and state notions; the ASM workbench[31] a comprehensive tool based on the ASM-SL language and equipped with a simulator, a GUI and model-checker; Asmeta[42] a meta model language along with a simulator, refinement prover, validation and model checking support; the ASML language [51] that provide a .NET based language supporting classes and interfaces; last but not least the CoreASM framework [37] a plugin based extensible framework supporting model simulation and debugging (see Section 2.4).

The experience gained with these tests made the ASM method a valid approach to tackle and manage the complexity of software engineering problems providing a way to define pseudo code over abstract data that is understandable even by non experts, executable for simulations prior to implementation, and mathematically precise for properties verification and correct implementation.

3.2 Lessons from the experience with ASMs (Shortcomings)

Although the ASM method has reached its maturation and the large set of cases in which it has been successfully applied establish it as a relevant formal method for system specification, there are some drawbacks for a fully satisfying adoption of the method.

The ASM language, from its initial definition of the Lipari guide, has been reshaped and extended multiple times in order to address uncomfortable modelling routines.

We now outline some of the major drawbacks of the method and how people have overcome them. Since for most of them we will provide an analysis and a solution in the following chapters, we now just go through them quickly to give a taste of the motivation that have driven us to look for a solution.

Guarded statement limitations

The first aspect we want to discuss is how the initial ASM defined in the Lipari guide has evolved with practical application of the method. The original definition, as reported in Section 3.1, consisted primarily of a set of guarded update instructions. The two major changes have been relaxation of places in which **if** statement may appear and on the body of the guarded statement. The body initially was composed only by update instructions that correspond to modification of the machine state. The guard was meant to discriminate *enabled* rules from the others. In order to have a more readable model, along with update instructions, other constructs were allowed to be part of the definition of rules body. Nested **if** statement have appeared inside the guarded body. On one end this choice allows the condition in the guard to be more readable and better presented. On the other hand the meaning of enabled rules, previously clearly identified by guard conditions, introduced a degree of confusion. From the point of view of the expressitivity is always possible to flatten nested if and go back to the initial ASM form.

For example the following guarded statement with nested if can be flattened into two guarded statement combining the inner condition with the top-level guard.

---

if a **then**
  if b **then**
    R_1
  else
    R_2

The result of such transformation is:

---

if $a \wedge b$ **then** R_1

if $a \wedge \neg b$ **then** R_2

The two versions are equivalent but the conditions become more complex. If the nesting level is deeper the readability of a model composed by this kind of rules deteriorates rapidly. Moreover **if** statements are suitable to describe the behaviour of a system but not its structure.

The second change to the language introduced a macro expansion mechanism. A macro is a map between a name and a set of transition rules. The macro may be parametric and, through a *macro expansion* process, each occurrence of the macro name is replaced by the corresponding set of transition rule. Macros represents the first form of modularity in ASMs for their capability to group and organise set of related rules.

An emblematic application of macros can be found in the model for the Java Virtual Machine (JVM) [84].

JVM ground model

execJava ≡
  execJava$_I$
  execJava$_C$
  execJava$_O$
  execJava$_E$
  execJava$_T$

execJava$_I$ ≡
  execJavaExp$_I$
  execJavaStm$_I$

execJava$_C$ ≡
  execJavaExp$_C$
  execJavaStm$_C$

execJava$_O$ ≡
  execJavaExp$_O$

execJava$_E$ ≡
  execJavaExp$_E$
  execJavaStm$_E$

execJava$_T$ ≡
  execJavaStm$_T$

The main rule (*execJava*) is a set of other macros named execJava$_x$ with $x \in \{I, C, O, E, T\}$ where the values of x represent *sub-models* respectively for the imperative (I), classes (C), object-orientation (O), exception handling (E) and concurrency via threads (T) portion of the Java language. The JBook model represents one of the most shining example of the ASM method success. Looking at the macro composing the execJava rule, they also are the parallel composition of other macros. This approach is effective when presenting an ASM model.

Nevertheless, macro expansion reveals the very nature of the model: a long block of **if** statements without structure. Building ASM models structuring them with macros requires *extreme care* since all the macros will be flattened and the growing complexity of systems requires several macro definitions that must ensure the absence of conflicts for the conditions inside their definition when composed in parallel with the others.

Unclear notion of module

As we already mentioned, a set of **if** statements easily models the behaviour of a system but not its structure. A notion of module, in time, has always been felt as needed. An example may be found in Mearelli's production cell model [67][68] where modules are presented as labels followed by a set of ASM transition rules. Unfortunately this approach does not scale well for complex systems.

The first real notion of module is present in [12]. The definition is meant to "syntactically structure large ASMs". The definition (see Chapter 2 for the whole definition) contains notion for *communication* between machines, module retrieval by *import* clause, and *visibility* selection of internal module functions. Unfortunately, at the best of my knowledge, nobody has never used this notion in a real case probably because the syntax to define and use the module is impractical.

Another form of modularization is presented again in [12] in the chapter dedicated to *structured ASMs*: TurboASM. In this case the focus is on how to modularize the computation. TurboASMs for sequential composition, iteration, recursion via submachines are provided. The behaviour of TurboASM, however, moves away from the original simplicity of the first ASM definition and does not represent structure.

Figure 3.1: Schmid's graphical notation for components

The work by Schmid in [81] gives another definition and graphical notation (see Figure 3.1) of module where the notion of component that requires inputs and provide output is defined for ASMs. The definition is justified, as Schmid writes, by the "difficult of dividing an ASM specification in smaller independent parts". For this reason he gives a definition that is tailored especially for its problem domain that is digital hardware circuits specification.

In [4], Asmundo et al. define two version of machine composition useful, for example, when the model can be seen as a device composed by different features. Two composition operators are defined but only for sequential ASMs.

Dausend in [29] brings the concepts of aspect oriented programming [61] into the specification world. One of the main benefit of aspect oriented approaches is to be able to isolate and define the so called crosscutting concerns of a problem that will be later *wove* (composed) together when executing the program (in this case when simulating the ASM machine). Unfortunately the model becomes scattered into several "aspects" making it less understandable and with a more obscure architectural structure.

Recently, ASM nets [19] gives another take to ASM modules. Here the aim is to specialise ASM, and in particular control state ASMs, for business process modelling (BPM). An explanation on how to tailor ASMs for BPM and a graphical notation (see Figure 3.2) are defined. ASM nets have been inspired by the IBM's Guard Stage Milestone (GSM) [57] approach. An ASM net is made of initial states (the stages) in which a set of enter conditions are tested, just like a control state ASM. If one of the enter conditions is true a machine M is executed until one of the exit conditions is true. An exit condition whose value is true sets the new stage from which the computation will continue. Wiring starting states and exit states compose the whole net.

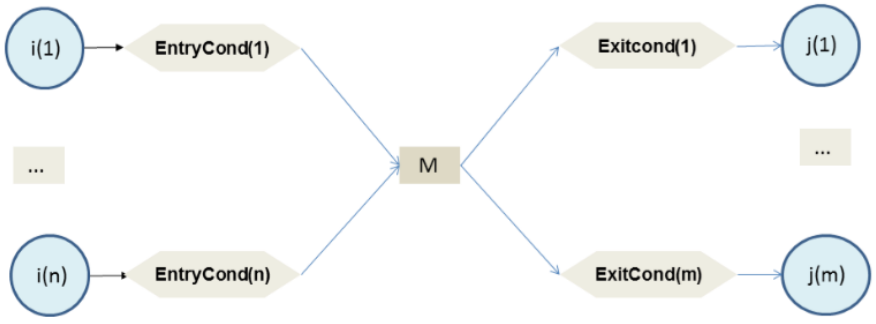


Figure 3.2: ASM nets diagram

Global state management

The state of an ASM is, by default, global. From any point in the model, every name of rule or function identify the same entity. Sometimes information hiding is a better approach for organising the model. Being sure that any modification in a controlled space will not affect other parts of the model is a desirable property. In fact there have been different attempt to manage the global state in order to achieve local state.

In the Lipari guide, the special function *self* was already present to identify the *private* state of an agent whose related location are usually implicitly parameterized with the value of *self*. This is a special case of location parameterization that is the more frequent approach to state partitioning. A clear example of state partitioning can be found in *control state* ASMs where some location, often called *mode*, *phase* or *stage*, is used to partition the behaviour of the machine. The manual management of *mode* locations is error prone and become difficult when the location is not parameterized by a single but multiple values. The approach in such cases requires to split the parameterization on multiple locations or to *carefully* assume a policy for it that each location occurrence must follow.

In [12] the **local** keyword appears. Its meaning is to provide local state inside a rule definition that, as a consequence, can have different incarnation of functions. For each of these local functions an *init* rule can be defined. This approach has the drawback of the restricting on the scope of these functions to the rule body, while is usually desirable to have a finer grain.

A more sophisticated version of state modification control could be found on *TurboASM*s where the rules behaviour collapse multiple steps of the machine into a single one discarding all the updates but one specific location updates (**result**).

Finally the ambient ASMs [17] introduce the support for "apparently any environment paradigm" through the definition of ambient expression, ambient ASMs isolate the desired portion of the state. As we will see in chapter 6, the expressiveness of ambients is undermined by their practical usage.

Inconvenient application of the method

Another aspect of modularity that can be improved in ASM relates to data representation modularity and in particular freely generated data types. In [35] a model for Virtual Filesystem Switch (VFS) using ASMs is provided. The authors however had to rely on the KIV[7] prover to axiomatized data types. Also in [60] the author state that "direct support for free data types in CoreASM would be preferable – which we have not addressed as it would require deep modifications, for example in the parser". These cases arise the need for freely generated data types support for ASMs and we will investigate them in this thesis.

Cooperative and reusable view of the method

The application of ASMs for producing models can be improved also from the point of view of the ground model creation process. Usually ground models are produced by a single person that has the complete view of the model or by a group of people that must work together to define it to ensure that the guards of the rules will not produce inconsistencies when all the macros are pulled together. Being able to divide the responsibilities and work independently only on portion of the specification can improve the process in the same way software project management tool do for programming languages packages. Another aspect is related to the *reuse* of specifications portion. Currently there is no proper support for *cooperative* work in the ASM method.

3.3 Proposal

The experience gained during the long history of the method creates a need for a comprehensive study on the modularity feature of ASMs and a consequent effort to provide a *language level solution* to such issues.

Analysing the history of the ASM method, we identified some features of ASMs that are not fully satisfying during the modelling process or some lacking support for other features that had as a consequence the choice to use other modelling methods (see [60]). Nevertheless the nature of ASMs with his abstraction freedom and simple core concepts make it really helpful in capturing system requirements, simulating prior to coding, understanding and reverse engineer already defined system, and verifying properties. Being *pseudo-code over abstract data* makes models readable for engineers and even for non experts. For this reason we believe that allowing such good features to scale with the ever growing systems complexity will allow the method to further be established and integrated in the software engineering process.

In this thesis we will introduce *novel constructs* to manage the complexity of software intensive systems with ASMs. Our goal is to give the instruments to achieve more manageable ASM models improving their modularity, clarity, readability and compactness of ASM specifications.

We will give the syntax and formal semantic for each construct in terms of the CoreASM[37] interpreter. CoreASM is a well established framework that supports the ASM method, it is mature and extensible via a plugin architecture that allows the interpreter to introduce new constructs into the framework seamlessly. This approach on the semantic definition make our modularity constructs ready for the integration with CoreASM, de facto enabling modularity feature to the ASM method. For this reason, we will try to be the less disruptive is possible with regard to the CoreASM interpreter in giving the semantic. When multiple solutions for the same problem will be available, we will choose the one that fits the most the CoreASM framework. The same approach will be applied to the mathematical foundation of ASMs: we will try not to rebuild from scratch the fundamental concepts of ASMs and whenever our approach could require such modifications, we will provide a translation that shows equivalence with the foundational definitions of ASMs.

We propose a language level solution to: module definition, modularization of computations, state management and data values representation and manipulation.

Chapter 4

Modularity of Computation

4.1 Control flow in ASMs

In this chapter we are interested in how to structure the control flow of ASM computations. In the history of ASM many different strategies for the execution of a set of statements have been developed.

Control flow is usually managed by programming languages using a program counter that points to the instruction to be executed, each construct changes the position of the program counter, and the sequence of such positions define the flow of a program. In ASMs the concept of control flow does not depend on a program counter, instead at each machine step a subset of rules that are part of the specification are selected to be performed. The aggregation of all the update sets produced by such rules determine the state evolution. The sequence of selected rules constitutes the flow of execution of an ASM specification.

The control flow can be altered in multiple ways. It is possible to act on the set of rules to be executed at each step, or on the number of times a rule must be executed. Another way to alter the control flow is to change the order in which a set of rules is executed. In the following sections we analyse these possibilities starting from the current status of the ASM language.

4.1.1 Rule execution conditions

The original ASM definition of the Lipari guide [45], defines the concept of enabled rules. The enabled rules of a computation are the set of rules that contribute to the next step in a run of an ASM applying the update instructions (assignment to the state locations) that make the state of the machine evolve. Guarded statements defined how to distinguish which rules are enabled. Guarded rules are *if* statements in the form **if** G **then** R where G is a boolean predicate on the current machine state called guard and R is a set of update instructions. An ASM model thus is a collection of guarded rules. At each step, the interpreter of the machine evaluates the guard and, if it is true, considers the corresponding update instructions (R) enabled.

Although the Lipari definition was already Turing complete, soon the practical application of ASMs have shown the need to change the expressiveness of the language. For this reason the update instructions have been allowed to contain not only update instructions but also macros and nested if statements. This choice allowed a better organisation of the execution conditions and presentation of the models (Control State ASMs are an notable example of this case). The downside of this approach is a dulled language semantic: the top-level *ifs* have a different semantic from the nested ifs. Enabled rules were selected by the interpreter and contribute to the computation with non empty update instructions sets. With the introduction of transition rules after the guard an enabled rule can produce an empty update set. So two forms of if statements have appeared. A top level *if* maintains the role of enabling or disabling rules, while the nested ifs introduced the possibility to produce empty update sets for enabled rules. An empty update rule can be generated, for example, by an inner if with false condition.

However the relation between top-level and nested ifs is really close. It is always possible, as we already mentioned in Section 3.2, to rewrite nested ifs and macros following the original semantic. Such transformation shows the equivalence of the more structured ASM models to the guarded rules version.

Translating nested if requires to compose complex conditions whose complexity depends on the degree of nesting of the if statements. This problem is already visible with a simple top-level if statement with a nested if:

---

if a **then**
  R_1
  if b **then**
    R_2
  else
    R_3

This would be translated into:

---

if a **then**
  R_1
if $a \wedge b$ **then**
  R_2
if $a \wedge \neg b$ **then**
  R_3

The readability of this version is worse and inspecting the model has become more difficult. Moreover, the translation to "simple" guarded rules introduces a great number of guards that depends on how many inner level ifs there are.

Although equivalent in behaviour, having the capability to structure the specification improves readability and understandably which are paramount in a specification language.

While, on one hand, the concept of enabled rules is related to the selection of when rules must be executed, on the other hand it is important to decide when to stop their execution. In the literature, ASM models do not contain the definition of a termination condition. Instead it is stipulated outside the specification. This has been the direct consequence of the run definition for ASMs. A computation is a, possibly infinite, sequence of moves that produces a sequence of update sets describing the evolution of the machine state. The termination conditions are usually encoded as functions of the update sets that are described outside the specification itself. Examples of termination conditions are: reaching an inconsistent update set, the same set of update is generated (a particular case is the empty update set), executing a fixed number of steps.

Nevertheless, the specification should contain every information about the problem domain it models, including termination conditions. The capability to specify the termination conditions for a whole computation (or sub-computation) improves the completeness of the ASM model and its clarity from the specification process point of view. So we intend to bring them back to the specification where they belong.

4.1.2 Rules composition

Knowing when to start and terminate the execution of a rule is not the only interesting operation on the control flow. Another distinctive operation on a set of rules is deciding which and how to execute them.

Given a set of rules it is possible to choose which rules should be executed. There are many ways of choosing among the rules. It can depend on the textual position, on some property of the state, on the computation itself. In the current ASM language there is no direct support for rule selection different from guarded statements. Of course it can be emulated with a level of encoding, and the Control State ASMs are an example of that. Writing:

```
if mode = 1 then
   R1
   mode := 2

if mode = 2 then
   R2
   mode := 3

...

if mode = n then
   Rn
   mode := 1
```

encodes in the *mode* location the selection of the R_i rules by a natural number. Controlling the selection in this case is achieved providing predicates on the *mode* location and managing its value. More control is possible if equality guards are replaced by more complex encoding or properties. Although feasible, this approach requires to maintain the encoding of properties making the process error prone and worsening the clarity of the specification.

The other operation on rule sets to control the flow is *how* to execute the selected rules. The ASM language provide various statements to control the composition of multiple rules. The most common example is the parallel block composition in which all the rules in the block are executed in parallel and the resulting update sets are joined to compose the final one. For this kind of flow composition the statement *par* and *parblock* have appeared in the language. Another type of flow composition is the sequentiality that allows two or more rules to be executed one after the other. Executing a set of rules in sequence can be performed in different ways. For example the *seq-next* and *seqblock* statement of Turbo ASM executes each rule in the rules block in sequence applying the intermediate updates to a provisional state until the last rule has been executed resulting in a single big step of the computation. Another sequential execution of the rules instead could be performed by applying the updates after the first rule has produced its update set and then, in the next step of the machine, continue the execution with the next rule and so on. Other possibilities could be found interesting for some systems.

4.1.3 Proposed solution

The ASM language provides a lot of different constructs to handle control flow management. Most of them are part of the Turbo ASM definitions where iteration, sequential execution and exception handling have been defined[1]. All the constructs have a specific reason for their introduction in the language but too many variants even for the same control flow specification have been generated. For example sequential and parallel execution have two definitions each (par, parblock, seq, seqblock). Is it possible to reduce all these construct to a single one? A positive answer to this question would simplify the semantic of several different execution strategies that have found various incarnation from Basic to Turbo ASMs. Moreover such single construct would also be helpful to organise and modularize the description of computations.

Here we are not implying that the current available constructs are not modular. The pseudo code of ASM statement is, of course, modular in the sense that it can be defined as the composition of such constructs. What we are trying to achieve is a uniform and unified view of the control flow in an attempt to simplify the definition of complex flows and to reduce future transition rule extensions to special cases of this unified view, and to make ASM specification more readable and complete.

Thus, although maintaining the semantic that each statement returns a set of updates, we intend to allow more latitude in the way they collect updates.

[1] See chapter 2 for the definitions

⟨*DoBlock*⟩ ::= '`do`' ⟨*ExecutionCondition*⟩ ⟨*ExecutionMethod*⟩ ⟨*SelectionMethod*⟩ '`enddo`'

⟨*ExecutionCondition*⟩ ::= '`if`' ⟨*Condition*⟩ | '`while`' ⟨*Condition*⟩ | '`until`' ⟨*Condition*⟩ | '`forever`'
| ⟨*Number*⟩ '`times`' | '`once`'

⟨*Condition*⟩ ::= ⟨*BooleanExpression*⟩ | '`noupdates`' | '`fixpoint`'

⟨*ExecutionMethod*⟩ ::= '`in sequence`' | '`stepwise`' | '`in parallel`'

⟨*SelectionMethod*⟩ ::= '`all of`' ⟨*Rules*⟩ ⟨*Reset*⟩?
| '`any of`' ⟨*Rules*⟩ ⟨*Reset*⟩?
| '`one of`' ⟨*Rules*⟩ ⟨*Reset*⟩?

⟨*Rules*⟩ ::= ⟨*Rule*⟩ | ⟨*Rule*⟩ ⟨*Rules*⟩

⟨*Reset*⟩ ::= '`reset on`' ⟨*Condition*⟩

Figure 4.1: do-block EBNF

In order to control the execution of a set of rules, the construct must provide a mechanism to define when the execution is allowed to be performed. After this check, the computation should proceed selecting how and which rules will contribute to the step. The construct would also contain a clear indication of when to terminate the execution. If putting these concepts together results in a easy to use compact and readable statement, the specification process will improve.

We propose a single ASM rule form to handle *selection*, *repetition*, *execution mode* and *termination conditions* for the execution of a set of ASM rules.

In detail, *selection* refers to the way a subset of all available rules is selected for execution (other available rules that are not selected are ignored and do not contribute updates to the execution step); *repetition* refers to the repeated execution of a set of rules over a single or multiple steps of the machine; *execution mode* refers to the way in which updates from executed rules are accumulated, and potentially applied to the state prior to the execution of further rules, and finally *termination condition* refers to the criterion for stopping the execution of a set of selected rules.

Since our attempt is to unify the semantic of every control flow construct for ASMs that has been defined till now, we would also like to define the construct in a way that is naturally extensible. This way it will not be yet another construct but can be the replacement for the well-known control flow statements and it will support future additions to the language as minor extension of the semantic we are going to provide.

Abiding to a long tradition in language design, we use the keyword **do** to introduce the control set of statements. We call the construct **do**-block.

In the following sections we define the syntax of the **do**-block and the intuitive semantic, followed by some examples and the full semantic in terms of the CoreASM interpreter [2].

4.2 Block rule syntax

For clarity, we start by presenting the syntax of the proposed block construct, so that semantics can be detailed later (see Section 4.6) with reference to syntactic clauses. Each clause parameterize some aspect of execution flow control.

The construct includes three clauses, two of which are optional:

[2] A variation of the **do**-block construct semantic has been developed independently and in parallel to our work at the University of Ulm by Michael Stegmaier. A paper about this work has been submitted to the ABZ 2016 conference.

1. a *repetition* clause, specifying when rules are enabled for execution (enter condition) and whether the block should be re-executed or not (exit condition). We have defined concrete clauses for standard control flow such as simple conditional (**if**), head- and tail-conditions for indefinite iteration (**while** and **until**), infinite repetition (**forever**) and two special conditions that predicate on the internals of the ASM computation step (**no updates** and **fixpoint**). Details are defined in Section 4.6.1. If the repetition clause is not provided, by default we assume that the repetition condition is **if** *true* (i.e., a single execution). Other forms of repetition can be defined; as an example, we suggest an implicit-counter determined iteration loop, and a guaranteed single-execution form.

2. an *execution mode* specifies how the selected rules have to be executed in a single iteration of the repetition loop. We define three execution modes:

   - **in parallel** executes all selected rules in the current state, accumulates the corresponding updates, and (provided they are consistent) the resulting update set is considered to be the updates generated by the given iteration of **do**-block. This mode coincides with the traditional (Lipari guide [45]) parallel execution of ASM rules, and is the default.
   - **in sequence** executes each of the selected rules in the provisional state resulting from applying the updates from the preceding rule to the state. The final update set is whatever is needed to update the initial state to the provisional state resulting from the execution of the last rule in the sequence. This mode coincides with the TurboASM definition of sequential execution (variously defined as **seq** − **next** , **seqblock** − **endseqblock** , etc.). In particular, the **in sequence** execution of a set of selected rules still constitutes a single step of the machine.
   - **stepwise** executes the selected rules, but each on a different (successive) step of the machine. Intuitively, the construct behaves as if an internal program counter directed the execution in sequence, at each step, of each of the selected rules.

3. the *selection* clause lists the rules that are candidates for execution, and specifies which subset has to be considered for actual execution on each iteration of the repetition clause. We define three selection policies; each is applied to a sequence of candidate rules and produces a sequence of selected rules.

   - **all of** $R_1 \dots R_n$ results in $[R_1 \dots R_n]$ being selected for execution;
   - **one of** $R_1 \dots R_n$ results in $[R_k]$ being selected for execution, with k non-deterministically chosen between 1 and n;
   - **any of** $R_1 \dots R_n$ results in $[R_{k_1} \dots R_{k_m}]$ being selected for execution, with $m \leq n$, and the result being a non-deterministically chosen, order-preserving, sparse sub-list of the source list.

     The selected rules can be changed repeating the selection process. To provide a finer grained control over the selected rules, we add a **reset on** condition for selection clauses. If the reset condition becomes true, a new list of rules is selected.

   The default is **all of** $R_1, \dots, R_n$ **reset on** $false$ with the meaning that the selection chooses all the rules and never resets.

4.3 Intuitive semantics

Intuitively, our **do**-block works as follows. On execution of the block, the execution condition is evaluated in the current state, resulting in a pair of boolean values (the enter-condition and the exit-condition). If the enter-condition is false, the block terminates resulting in an empty update set. Otherwise, a sequence of rules from the candidates list is selected, according to the selection clause. We call these the *enabled* rules.

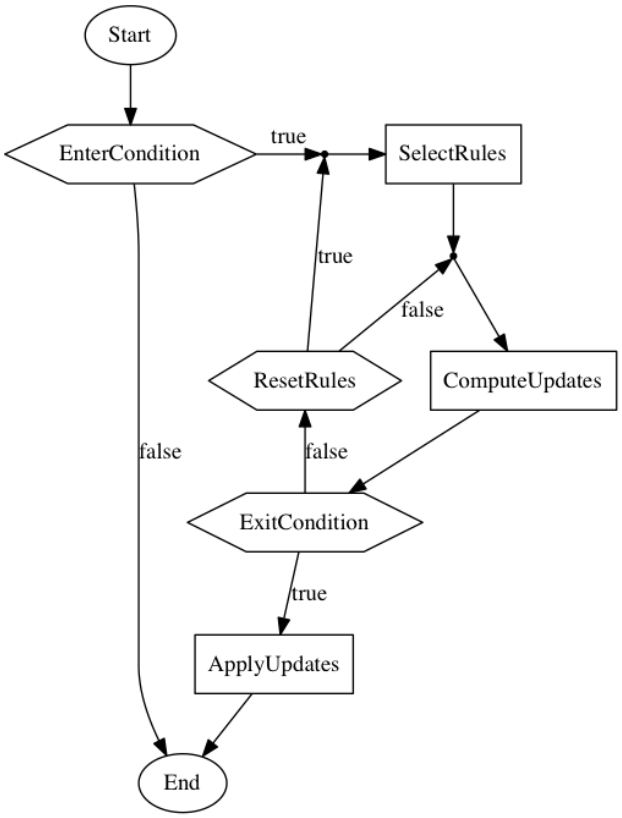


Figure 4.2: do-block semantic diagram

Execution then proceeds by computing the updates from the enabled rules, by executing all of them in parallel or in sequence, or by executing one of them (the "next" one according to the internal program counter) in case of stepwise execution.

The produced updates[3] are then provisionally applied to the state, and the execution condition is evaluated again (in the modified state). If the exit-condition is true, the execution of the **do**-block is complete, resulting in the set of updates needed to transform the initial state (the one in effect upon the initial execution of the **do**-block) into the final provisional state. Otherwise, the reset condition is evaluated and the execution of the enabled rules is repeated, in the current provisional state. The diagram in Figure 4.2 shows the control state ASM version of the **do**-block behaviour.

In the following section we show some usage examples of the **do**-block construct. For the formal definition of the semantic refer to Section 4.6

4.4 Examples

In this section we present some examples for the **do**-block. The specifications proposed as examples compare the new construct with the classic ASM language. With these examples we want to show some concrete cases of **do**-block usage and how the proposed transition rule can simplify and accelerate the modelling process. Moreover the resulting model improves two of the key aspects of ground models: simplicity and conciseness.

[3]Technically: update instructions, but we can abstract here from the details of the aggregation process.

4.4.1 Round-robin DNS

In this example we specify a simple round-robin Domain Name System (DNS) server. A DNS servers implement a translation service that maps resource names (usually domain names) to internet protocol (IP) addresses. A round-robin DNS provides, for a single resource, multiple IP addresses from a list. They are often used for load balancing the resource request splitting them among, for example, multiple servers. The procedure of choosing the IP from the available list is not standardised but, in round-robin approach, each DNS request permutes the list of IPs in circular order.

When a DNS request arrives the condition *hasPendingRequests* enables the **do**-block that **stepwise** performs one of the corresponding IPs.

doBlock

```
RoundRobinDNS ≡
    do if hasPendingRequests stepwise
      AnswerReqWith(ip1)
      AnswerReqWith(ip2)
      AnswerReqWith(ip3)
    doend
```

A possible version of the same example with classic ASMs, requires to encode the stepwise behaviour with a function *nextIp* that must be initialised. The **do**-block rule avoids the initialisation step for the set of available IPs.

asm

```
init ≡
    ...
    currentIp := ip1
    nextIp(ip1) := ip2
    nextIp(ip2) := ip3
    nextIp(ip3) := ip1
    ...

RoundRobinDNS ≡
    if hasPendingRequests then
      AnswerReqWith(nextIp(currentIp))
      currentIp := nextIp(currentIp)
```

4.4.2 Book

In this example we specify the process of buying a book online. A customer can select books to buy adding them to the shopping cart. Whenever the customer decides to pay for the book, he checks the cart, then inserts payment info and finally terminates the payment with the checkout. The customer may end the shopping *anytime*. The process can be described as a continuous choice of one of the possible actions the customer may perform that terminates when the session ends (the payment or an explicit termination are performed). The checkout procedure is a sequential process that requires each step to be performed one after the other. We have chosen to define it as a stepwise execution strategy because we want to leave open the possibility to end the session before payment. If the checkout procedure is meant to be non interruptible, changing *stepwise* with *in sequence* is enough.

doblock

```
BookShoppingProcess ≡
  do stepwise all of
    do until customerEndsSession one of
      AddBookToCart
      do stepwise all of
        CheckCart
        InsertPaymentInfo
        DoCheckout
      doend
    doend
    EndSession
  doend
```

In this example we highlight that a one-to-one translation from the **do**-block to the classic version for this specification is not straightforward. The rule selector **one of** ensure that exactly one of the inner rules will be enabled at each step of the execution. In the classic version we needed to prioritise the check for some of the operation so that only one of the sub-processes will produce updates in order to avoid possible inconsistent updates. Moreover the payment process is encoded by a control state ASM that reduces the readability.

asm

```
BookShoppingProcess ≡
  if customerWantsToEndShopping then
    EndSession
  else
    if customerWantsToBuyBook then
      AddBookToCart
    else
      if mode = Idle ∧ ReadyToPay then
        mode := CheckingCart
      if mode = CheckingCart then
        CheckCart
        mode := InsertingInfo
      if mode = InsertingInfo then
        InsertPaymentInfo
        mode := DoingCheckout
      if mode = DoingCheckout then
        DoCheckOut
        mode := Idle
```

Another way of specifying such behaviour (with classic ASMs) could have been to encode the possible choices, for example with an enumeration, and non deterministically choose among them guarding the execution of the related rules. We will show such approach in the example of Section 4.4.4.

4.4.3 AJAX Call

In this example we give a specification for a simplified AJAX call. AJAX stands for asynchronous Javascript and XML and is a collection of web development techniques to create asynchronous web applications. AJAX is based on the capability to send and receive data from a web page independently from the display process. AJAX calls provide the data interchange mechanism.

An AJAX call is composed by three steps: sending a request to a web server, waiting for the answer and processing the received data when the answer arrives. The natural way to describe this behaviour with the **do**-block is to put the three steps performed by a *stepwise* execution strategy.

After the SendRequest rule execution, we want to show a waiting animation (usually called

trobber) until an answer is received (its state is *ready*). The until execution condition handles this situation. Finally DisplayData processes the received data displaying the request answer.

doblock

```
HandleAjaxCall ≡
  do stepwise
    SendRequest
    do until RequestState = Ready
      ShowThrobber
    doend
    DisplayData
  doend
```

To describe the same example with classic ASMs, the natural way is to define a Control State ASM whose mode represents the status of the request. The *mode* location contains the information about the request status and must be initialised somewhere outside the HandleAjaxCall rule. The **do**-block version removes the need of initialisation since the mode is implicit in the composition of do statements providing less encoding and improved compactness and readability.

asm

```
...
mode := SendingRequest
...
HandleAjaxCall ≡
  if mode = SendingRequest then
    SendRequest
    mode := WaitingReadyState
  if mode = WaitingReadyState then
    if RequestState ≠ Ready then
      ShowThrobber
    else
      mode := ReadyToDisplay
  if mode = ReadyToDisplay then
    DisplayData
```

4.4.4 Mouse Input Simulator

Another example that shows how the **do**-block rule improves compactness and readability of models is the specification of a mouse input simulator. The model must provide a simulation of the events generated by an hardware mouse.

The rule MouseInputSimulator emulates the possible mouse inputs. A mouse can be left- or right-clicked, moved and its wheel can be shifted up or down. Each of this inputs may occur simultaneously but the wheel shift that cannot be moved up and down at the same time.

doblock

```
MouseInputEmulator ≡
  do any of
    MouseLeftClick
    MouseRightClick
    do one of
      MouseWheelUp
      MouseWheelDown
    doend
    MouseMove
  doend
```

Modelling the mouse simulator is simply the execution of *any of* the possible events with the

exception of the wheel inputs. Since they are mutual exclusive we group them inside another **do**-block that selects only one of the two possibilities.

The classic ASM version of the input simulator instead requires to encode the list of inputs and to choose a subset of them explicitly. On such subset a *forall* statement is in charge of executing the rule that fires the corresponding event. The mouse wheel events requires a double encoding to discriminate the event cases similarly to the other inputs.

asm

```
MouseInputEmulator ≡
  let Inputs = {LClick, RClick, Wheel, Move} in
  choose I with I ⊆ Inputs do
    forall x in I do
      case x of
        LClick :
          MouseLeftClick
        RClick :
          MouseRightClick
        Move :
          MouseMove
        Wheel :
          choose w in {Up, Down} do
            if w = Up then
              MouseWheelUp
            else
              MouseWheelDown
```

4.4.5 Cellular Automaton

This example implements the state evolution of a cellular automaton. In particular the following specifications implements rule 44 of the Wolfram code for elementary cellular automata. The computation terminates when the updates produced does not change the state (a fixpoint has been reached). The automaton consists of a set of cells. Each cell can be in the alive and dead state and a set of neighbours denotes the relation of adjacency with other cells. The system evolves based on the initial cell configuration and on some rules that take into account the neighbourhood state to determine a cell status change.

We assume that the functions *leftIsAlive*, *rightisAlive* and *selfIsAlive* are defined and their evaluation result in a boolean value representing the status of the adjacent cells and of the parameter cell.

The **do**-block allows to specify compactly the initialisation and the termination condition for the automaton. The *once* execution condition performs the inner rules just one time and is useful for initialisation. The fixpoint condition instead allows to assess a condition on the state of the computation and determine when termination occurs.

doblock

```
CellularAutomata ≡
  do until fixpoint
    do once
     forall c ∈ Cells do
       state(c) := pick x in {Alive, Dead}
    doend
    forall c ∈ Cells do
      if (leftIsAlive(c) and rightIsAlive(c)) or
        (leftIsAlive(c) and not rigthIsAlive(c)
                and selfIsAlive(c)) then
          ChangeCellState(c)
 doend
```

In classic ASMs the initial state has to be stipulated *outside* the specification. CoreASM improves on this situation by introducing a special *init* rule that is performed first at the beginning of the specification execution and then is disabled in the succeeding steps. The init rule contains all the update instructions that initialise the machine state and sets the program of the running agents. An additional encoding is required to check the state evolution in the form of *stateIsChanged* location. This encoding, besides being unnecessary, requires to be managed and worsen the readability of the specification.

asm

```
...
Init ≡
  stateIsChanged := true
  program(self) := @CellularAutomata
...
CellularAutomata ≡
  while stateIsChanged do
    seq
      stateIsChanged := false
    next
      forall c ∈ Cells do
        if (leftIsAlive and rightIsAlive) or
          (leftIsAlive and not rigthIsAlive and
                      selfIsAlive) then
            ChangeCellState(c)
            stateIsChanged := true
```

4.5 Formal semantics

In this section we present the formal semantics for the **do**-block construct by providing a calculi for the transition rules. Each inference rule derives the update set that the transition rule yields. Since the combination of the various execution conditions would generate a large number of cases, we summarise all the execution condition by describing only their evaluation result as a pair (ϕ_1, ϕ_2) where the ϕ_1 is the formula representing the entering condition and the ϕ_2 is the formula representing the exit condition. For each case of the execution conditions we will provide an interpreter semantic in section 4.6. For presentation reasons we separate the **do**-block execution strategy and selection clause parts from the execution conditions. For this reason, the rule R presented in section 4.5.1 are constrained to the rules described in the sections 4.5.2, 4.5.3, 4.5.4.

4.5.1 Do-block

When the enter condition φ_1 evaluates to $false$, the entire rule returns an empty update set.

$$\frac{[\![\varphi_1]\!]^S_\zeta = false}{[\![\textbf{do } (\varphi_1, \varphi_2)\ ^\alpha R \textbf{ doend}]\!]^S_\zeta = \emptyset} \quad (1)$$

If the enter condition φ_1 evaluates to *true*, the rule R is evaluated yielding the update set U. If the evaluation of the exit condition firing U is *true*, U is the resulting update set. Otherwise, the block is evaluated again in a new state that is the result of firing U; the resulting update set V is composed with U to get the final update set.

$$\frac{[\![\varphi_1]\!]^S_\zeta = true \quad [\![^\alpha R]\!]^S_\zeta = U \quad [\![\varphi_2]\!]^{S+U}_\zeta = true}{[\![\textbf{do } (\varphi_1, \varphi_2)\ ^\alpha R \textbf{ doend}]\!]^S_\zeta = U} \quad (2)$$

$$\frac{[\![\varphi_1]\!]^S_\zeta = true \quad [\![^\alpha R]\!]^S_\zeta = U \quad [\![\varphi_2]\!]^{S+U}_\zeta = false \quad [\![\textbf{do } (\varphi_1, \varphi_2)\ ^\alpha R \textbf{ doend}]\!]^{S+U}_\zeta = V}{[\![\textbf{do } (\varphi_1, \varphi_2)\ ^\alpha R \textbf{ doend}]\!]^S_\zeta = U \oplus V} \quad (3)$$

4.5.2 In parallel

In this section we define the semantics for the execution strategy *in parallel*. If the selection clause is *all of*, the execution of this strategy is the union of the updates produced by the rules $R_1, \ldots, R_n$ in the initial state.

$$\frac{[\![R_1]\!]^S_\zeta = U_1 \quad \ldots \quad [\![R_n]\!]^S_\zeta = U_n}{[\![\textbf{in parallel all of } R_1 \ldots R_n]\!]^S_\zeta = \bigcup_{i \in \{1,\ldots,n\}} U_i}$$

Evaluating in parallel a set of rules with the selection clause *one of* requires the selection of a single rule picked in the set $R_1, \ldots, R_n$. The function *selectedRule* stores such information. Initially the value of *selectedRule* is undefined for every rule identifier α, in this case an R_i is selected and evaluated yielding the U update set. The resulting update set for the whole rule must update the selected rule for the current instance. If the reset condition φ evaluates to *true* in the state after firing U, a new rule is selected.

$$\frac{[\![R_i]\!]^S_\zeta = U \quad i \in \{1, \ldots, n\} \quad [\![selectedRule(\alpha)]\!]^S_\zeta = \texttt{undef} \quad [\![\varphi]\!]^{S+U}_\zeta = false}{[\![^\alpha\textbf{in parallel one of } R_1 \ldots R_n \textbf{ reset on } \varphi]\!]^S_\zeta = U \cup \{((selectedRule, \alpha), R_i)\}} \quad (1)$$

$$\frac{[\![R_i]\!]^S_\zeta = U \quad i, j \in \{1, \ldots, n\} \quad [\![selectedRule(\alpha)]\!]^S_\zeta = \texttt{undef} \quad [\![\varphi]\!]^{S+U}_\zeta = true}{[\![^\alpha\textbf{in parallel one of } R_1 \ldots R_n \textbf{ reset on } \varphi]\!]^S_\zeta = U \cup \{((selectedRule, \alpha), R_j)\}} \quad (2)$$

$$\frac{[\![selectedRule(\alpha)]\!]^S_\zeta = R \quad [\![R]\!]^S_\zeta = U \quad [\![\varphi]\!]^{S+U}_\zeta = false}{[\![^\alpha\textbf{in parallel one of } R_1 \ldots R_n \textbf{ reset on } \varphi]\!]^S_\zeta = U} \quad (3)$$

$$\frac{[\![R]\!]^S_\zeta = U \quad i \in \{1, \ldots, n\} \quad [\![selectedRule(\alpha)]\!]^S_\zeta = R \quad [\![\varphi]\!]^{S+U}_\zeta = true}{[\![^\alpha\textbf{in parallel one of } R_1 \ldots R_n \textbf{ reset on } \varphi]\!]^S_\zeta = U \cup \{((selectedRule, \alpha), R_i)\}} \quad (4)$$

Like the *one of* clause, also the evaluation of a subset of rules as a result of the *any of* clause needs to store the information about the selected subset. In the following inference rules, the symbol $\sqsubseteq^*$ denotes the sub-sequence operator.

$$\frac{[\![R_{n_1}]\!]^S_\zeta = U_{n_1} \quad \dots \quad [\![R_{n_h}]\!]^S_\zeta = U_{n_h} \quad \langle n_1, \dots, n_h\rangle \sqsubseteq^* \langle 1, \dots, n\rangle \quad [\![selectedRule(\alpha)]\!]^S_\zeta = \texttt{undef} \; [\![\varphi]\!]^{S+U_{n_1}+\dots+U_{n_h}}_\zeta = false}{[\![^\alpha \textbf{in parallel any of } R_1 \dots R_n \textbf{ reset on } \varphi]\!]^S_\zeta = \bigcup_{i=n_1}^{n_h} U_i \cup \{((selectedRule, \alpha), \langle R_{n_1}, \dots, R_{n_h}\rangle)\}} \quad (1)$$

$$\frac{[\![R_{n_1}]\!]^S_\zeta = U_{n_1} \quad \dots \quad [\![R_{n_h}]\!]^S_\zeta = U_{n_h} \quad \langle n_1, \dots, n_h\rangle \sqsubseteq^* \langle 1, \dots, n\rangle \quad [\![selectedRule(\alpha)]\!]^S_\zeta = \texttt{undef} \; [\![\varphi]\!]^{S+U_{n_1}+\dots+U_{n_h}}_\zeta = true \; \langle n'_1, \dots, n'_h\rangle \sqsubseteq^* \langle 1, \dots, n\rangle}{[\![^\alpha \textbf{in parallel any of } R_1 \dots R_n \textbf{ reset on } \varphi]\!]^S_\zeta = \bigcup_{i=n_1}^{n_h} U_i \cup \{(selectedRule, \alpha), \langle R_{n'_1}, \dots, R_{n'_h}\rangle)\}} \quad (2)$$

$$\frac{[\![R_{n_1}]\!]^S_\zeta = U_{n_1} \quad \dots \quad [\![R_{n_h}]\!]^S_\zeta = U_{n_h} \quad [\![selectedRule(\alpha)]\!]^S_\zeta = \langle R_{n_1}, \dots, R_{n_h}\rangle \; [\![\varphi]\!]^{S+U_{n_1}+\dots+U_{n_h}}_\zeta = false}{[\![^\alpha \textbf{in parallel any of } R_1 \dots R_n \textbf{ reset on } \varphi]\!]^S_\zeta = \bigcup_{i=n_1}^{n_h} U_i} \quad (3)$$

$$\frac{[\![selectedRule(\alpha)]\!]^S_\zeta = \langle R_{n_1}, \dots, R_{n_h}\rangle \quad [\![R_{n_1}]\!]^S_\zeta = U_{n_1} \quad \dots \quad [\![R_{n_h}]\!]^S_\zeta = U_{n_h} \quad [\![\varphi]\!]^{S+U_{n_1}+\dots+U_{n_h}}_\zeta = true \; \langle n'_1, \dots, n'_h\rangle \sqsubseteq^* \langle 1, \dots, n\rangle}{[\![^\alpha \textbf{in parallel any of } R_1 \dots R_n \textbf{ reset on } \varphi]\!]^S_\zeta = \bigcup_{i=n_1}^{n_h} U_i \cup \{(selectedRule, \alpha), \langle R_{n'_1}, \dots, R_{n'_h}\rangle)\}} \quad (4)$$

4.5.3 In sequence

In this section we define the semantics for the execution strategy *in sequence*. Executing all the rules $R_1, \dots, R_n$ sequentially means to evaluate each rule R_i, starting from the initial state S, in the state that results from firing the previous update set U_{i-1}. The update set that the *in sequence* rule yields will be the composition of such updates.

$$\frac{[\![R_1]\!]^S_\zeta = U_1 \quad \dots \quad [\![R_n]\!]^{S+U_1+\dots+U_{n-1}}_\zeta = U_n}{[\![\textbf{in sequence all of } R_1 \dots R_n]\!]^S_\zeta = \bigoplus_{i\in\{1,\dots,n\}} U_i}$$

If the *one of* selection clause is applied to a set of rules, the rule evaluation requires to evaluate one rule R_i in the current state. The selected rule is stored inside the function *selectedRule* if the reset condition φ evaluates to $false$. Otherwise a new rule must be selected. The final update set for the rule is the union of U, the update set of the evaluated rule, and the update for the location holding the rule selection.

$$\frac{[\![R_i]\!]^S_\zeta = U \quad i \in \{1, \dots, n\} \quad [\![selectedRule(\alpha)]\!]^S_\zeta = \texttt{undef} \quad [\![\varphi]\!]^{S+U}_\zeta = false}{[\![^\alpha \textbf{in sequence one of } R_1 \dots R_n \textbf{ reset on } \varphi]\!]^S_\zeta = U \cup \{((selectedRule, \alpha), R_i)\}} \quad (1)$$

$$\frac{[\![R_i]\!]^S_\zeta = U \quad i, j \in \{1, \dots, n\} \quad [\![selectedRule(\alpha)]\!]^S_\zeta = \texttt{undef} \quad [\![\varphi]\!]^{S+U}_\zeta = true}{[\![^\alpha \textbf{in sequence one of } R_1 \dots R_n \textbf{ reset on } \varphi]\!]^S_\zeta = U \cup \{((selectedRule, \alpha), R_j)\}} \quad (2)$$

$$\frac{[\![selectedRule(\alpha)]\!]^S_\zeta = R \quad [\![R]\!]^S_\zeta = U \quad [\![\varphi]\!]^{S+U}_\zeta = false}{[\![^\alpha \textbf{in sequence one of } R_1 \dots R_n \textbf{ reset on } \varphi]\!]^S_\zeta = U} \quad (3)$$

$$\frac{[\![R]\!]^S_\zeta = U \quad i \in \{1,\ldots,n\} \quad [\![selectedRule(\alpha)]\!]^S_\zeta = R \quad [\![\varphi]\!]^{S+U}_\zeta = true}{[\![^\alpha \textbf{in sequence one of } R_1 \ldots R_n \textbf{ reset on } \varphi]\!]^S_\zeta = U \cup \{((selectedRule, \alpha), R_i)\}} \quad (4)$$

The *any of* selection clause, identify a subset of rules to be executed. Such subset is stored in the *selectedRule* function just like the *one of* version of this **do**-block rule version. Similarly to the *all of* version for the sequential execution of rules, all the rules in the selected subset are executed in the appropriate state and their update sets are composed to get the final update set.

$$\frac{\begin{array}{c}[\![R_{n_1}]\!]^S_\zeta = U_{n_1} \quad \ldots \quad [\![R_{n_h}]\!]^{S+U_{n_1}+\cdots+U_{n_h-1}}_\zeta = U_{n_h} \quad \langle n_1,\ldots,n_h\rangle \sqsubseteq^* \langle 1,\ldots,n\rangle \\ [\![selectedRule(\alpha)]\!]^S_\zeta = \texttt{undef} \ [\![\varphi]\!]^{S+U_{n_1}+\cdots+U_{n_h}}_\zeta = false\end{array}}{[\![^\alpha \textbf{in sequence any of } R_1 \ldots R_n \textbf{ reset on } \varphi]\!]^S_\zeta = \bigoplus_{i=n_1}^{n_h} U_i \cup \{((selectedRule, \alpha), \langle R_{n_1},\ldots,R_{n_h}\rangle)\}} \quad (1)$$

$$\frac{\begin{array}{c}[\![R_{n_1}]\!]^S_\zeta = U_{n_1} \quad \ldots \quad [\![R_{n_h}]\!]^{S+U_{n_1}+\cdots+U_{n_1-1}}_\zeta = U_{n_h} \quad \langle n_1,\ldots,n_h\rangle \sqsubseteq^* \langle 1,\ldots,n\rangle \\ [\![selectedRule(\alpha)]\!]^S_\zeta = \texttt{undef} \ [\![\varphi]\!]^{S+U_{n_1}+\cdots+U_{n_h}}_\zeta = true \ \langle n'_1,\ldots,n'_h\rangle \sqsubseteq^* \langle 1,\ldots,n\rangle\end{array}}{[\![^\alpha \textbf{in sequence any of } R_1 \ldots R_n \textbf{ reset on } \varphi]\!]^S_\zeta = \bigoplus_{i=n_1}^{n_h} U_i \cup \{(selectedRule, \alpha), \langle R_{n'_1},\ldots,R_{n'_h}\rangle)\}} \quad (2)$$

$$\frac{\begin{array}{c}[\![R_{n_1}]\!]^S_\zeta = U_{n_1} \quad \ldots \quad [\![R_{n_h}]\!]^{S+U_{n_1}+\cdots+U_{n_h-1}}_\zeta = U_{n_h} \\ [\![selectedRule(\alpha)]\!]^S_\zeta = \langle R_{n_1},\ldots,R_{n_h}\rangle \ [\![\varphi]\!]^{S+U_{n_1}+\cdots+U_{n_h}}_\zeta = false\end{array}}{[\![^\alpha \textbf{in sequence any of } R_1 \ldots R_n \textbf{ reset on } \varphi]\!]^S_\zeta = \bigoplus_{i=n_1}^{n_h} U_i} \quad (3)$$

$$\frac{\begin{array}{c}[\![selectedRule(\alpha)]\!]^S_\zeta = \langle R_{n_1},\ldots,R_{n_h}\rangle \quad [\![R_{n_1}]\!]^S_\zeta = U_{n_1} \quad \ldots \quad [\![R_{n_h}]\!]^{S+U_{n_1}+\cdots+U_{n_h-1}}_\zeta = U_{n_h} \\ [\![\varphi]\!]^{S+U_{n_1}+\cdots+U_{n_h}}_\zeta = true \ \langle n'_1,\ldots,n'_h\rangle \sqsubseteq^* \langle 1,\ldots,n\rangle\end{array}}{[\![^\alpha \textbf{in sequence any of } R_1 \ldots R_n \textbf{ reset on } \varphi]\!]^S_\zeta = \bigoplus_{i=n_1}^{n_h} U_i \cup \{(selectedRule, \alpha), \langle R_{n'_1},\ldots,R_{n'_h}\rangle)\}} \quad (4)$$

4.5.4 Stepwise

In this section we define the semantic of the execution strategy *stepwise*. Given a list of rules $R_1,\ldots,R_n$, this strategy starts executing the first rule R_1. Each new evaluation of this **do**-block form, selects the next rule in the list (following the textual order) and evaluates it. In order to remember which rule to execute in the current evaluation step, the function *lpc* stores this information and behaves as a local program counter with respect to the rule identifier.

$$\frac{i = \begin{cases} 0 & \text{if } [\![lpc(\alpha)]\!]^S_\zeta = \texttt{undef} \\ [\![lpc(\alpha)]\!]^S_\zeta & \text{otherwise}\end{cases} \qquad [\![R_i]\!]^S_\zeta = U}{[\![^\alpha \textbf{stepwise all of } R_0,\ldots,R_{n-1}]\!]^S_\zeta = U \cup \{((lpc, \alpha), i+1 \pmod n)\}}$$

Similarly to the parallel and sequential execution strategies, the *one of* and *all of* selection clauses need to store also the information about the selected rules. For this purpose we use the function *selectedRule* with the rule identifier α as parameter.

$$\frac{R = \begin{cases} R_i, i \in \{0,\ldots,n-1\} & \text{if } [\![selectedRule(\alpha)]\!]^S_\zeta = \texttt{undef} \\ [\![selectedRule(\alpha)]\!]^S_\zeta & \text{otherwise}\end{cases} \qquad [\![R]\!]^S_\zeta = U \quad [\![\varphi]\!]^{S+U}_\zeta = false}{[\![^\alpha \textbf{stepwise one of } R_0,\ldots,R_{n-1} \textbf{ reset on } \varphi]\!]^S_\zeta = U \cup \{((selectedRule, \alpha), R)\}} \quad (1)$$

$$\frac{\begin{array}{c}[\![R]\!]^S_\zeta = U \quad [\![\varphi]\!]^{S+U}_\zeta = true \quad j \in \{0, n-1\} \\ R = \begin{cases} R_i, i \in \{0, \ldots, n-1\} & \text{if } [\![selectedRule(\alpha)]\!]^S_\zeta = \texttt{undef} \\ [\![selectedRule(\alpha)]\!]^S_\zeta & \text{otherwise} \end{cases}\end{array}}{[\![^\alpha \textbf{stepwise one of } R_0, \ldots, R_{n-1} \textbf{ reset on } \varphi]\!]^S_\zeta = U \cup \{((selectedRule, \alpha), R_j)\}} \quad (2)$$

$$\frac{\begin{array}{c}[\![selectedRule(\alpha)]\!]^S_\zeta = \texttt{undef} \quad \langle R_{n_0}, \ldots, R_{n_{h-1}}\rangle \sqsubseteq^* \langle R_0, \ldots, R_{n-1}\rangle \\ r = \langle R_{n_0}, \ldots, R_{n_{h-1}}\rangle \\ [\![R_{n_0}]\!]^S_\zeta = U \quad [\![\varphi]\!]^{S+U}_\zeta = false\end{array}}{[\![^\alpha \textbf{stepwise any of } R_0, \ldots, R_{n-1} \textbf{ reset on } \varphi]\!]^S_\zeta = U \cup \{((selectedRule, \alpha), r), ((lpc, \alpha), 1 \pmod h)\}} \quad (1)$$

$$\frac{\begin{array}{cc}\langle R_{n_0}, \ldots, R_{n_{h-1}}\rangle \sqsubseteq^* \langle R_0, \ldots, R_{n-1}\rangle & \langle R_{n'_0}, \ldots, R_{n'_{k-1}}\rangle \sqsubseteq^* \langle R_0, \ldots, R_{n-1}\rangle \\ r = \langle R_{n_0}, \ldots, R_{n_{h-1}}\rangle & r' = \langle R_{n'_0}, \ldots, R_{n'_{k-1}}\rangle \\ [\![R_{n_0}]\!]^S_\zeta = U \quad [\![\varphi]\!]^{S+U}_\zeta = true \quad [\![selectedRule(\alpha)]\!]^S_\zeta = \texttt{undef} & \end{array}}{[\![^\alpha \textbf{stepwise any of } R_0, \ldots, R_{n-1} \textbf{ reset on } \varphi]\!]^S_\zeta = U \cup \{((selectedRule, \alpha), r'), ((lpc, \alpha), 0)\}} \quad (2)$$

$$\frac{[\![selectedRule(\alpha)]\!]^S_\zeta = \langle R_{n_0}, \ldots, R_{n_{h-1}}\rangle \quad [\![lpc(\alpha)]\!]^S_\zeta = i \quad [\![R_{n_i}]\!]^S_\zeta = U \quad [\![\varphi]\!]^{S+U}_\zeta = false}{[\![^\alpha \textbf{stepwise any of } R_0, \ldots, R_{n-1} \textbf{ reset on } \varphi]\!]^S_\zeta = U \cup \{((lpc, \alpha), i+1 \pmod h)\}} \quad (3)$$

$$\frac{\begin{array}{c}[\![selectedRule(\alpha)]\!]^S_\zeta = \langle R_{n_0}, \ldots, R_{n_h}\rangle \quad [\![lpc(\alpha)]\!]^S_\zeta = i \quad [\![R_{n_i}]\!]^S_\zeta = U \quad [\![\varphi]\!]^{S+U}_\zeta = true \\ \langle R_{n'_0}, \ldots, R_{n'_{k-1}}\rangle \sqsubseteq^* \langle R_0, \ldots, R_{n-1}\rangle \\ r = \langle R_{n'_0}, \ldots, R_{n'_{k-1}}\rangle\end{array}}{[\![^\alpha \textbf{stepwise any of } R_0, \ldots, R_{n-1} \textbf{ reset on } \varphi]\!]^S_\zeta = U \cup \{((lpc, \alpha), 0), ((selectedRule, \alpha), r)\}} \quad (4)$$

4.6 Interpreter semantics

In this section, we present the formal semantics of our proposed construct by providing an ASM specification for an interpreter. The rules presented here integrate with the CoreASM language [37], both at the semantics level, and in implementation (through CoreASM's plug-in architecture).

The semantic of the **do**-block is defined in terms of the result of the evaluation of its clauses. For clarity and readability we have decided to present the clauses semantic separately from the general **do**-block semantic.

The execution strategy clause is denoted by the additional notation ${}^\alpha s$ that is to be considered part of the notation defined in Section 2.4.2.

Such notation is an abbreviation for the pattern notation whose condition part is $token(\alpha) = \mathsf{Id} \wedge token(\alpha) \in \{\mathsf{stepwise}, \mathsf{inparallel}, \mathsf{insequence}\}$ and the action part is $token(\alpha)$. This way we abstract the **do**-block semantic from the execution strategy. We will address separately the logic of each strategy providing an instance of the parametric rule $\mathsf{ComputeUpdates}_{token(\alpha)}$.

do-block semantic

$(\!|\ \mathbf{do}^{\alpha}\square^{\beta}s^{\gamma}\square\ \mathbf{doend}\ |\!) \rightarrow$
$\quad pos := \alpha$
$\quad rulePos(\alpha) := pos$

$(\!|\ \mathbf{do}^{\alpha}v^{\beta}s^{\gamma}\square\ \mathbf{doend}\ |\!) \rightarrow$ **if** $enterCond(^{\alpha}v)$ **then**
$\quad pos := \gamma$
$\quad rulePos(\gamma) := pos$
$\quad composedUpdates(pos) := \{\}$
$\quad$ PushState
else
$\quad [\![pos]\!] := (\mathtt{undef}, \{\}, \mathtt{undef})$
$\quad lastUpdates(pos) := composedUpdates(pos)$

$(\!|\ \mathbf{do}^{\alpha}v^{\beta}s^{\gamma}v\ \mathbf{doend}\ |\!) \rightarrow$ **if** $currUpdates(pos) = \mathtt{undef}$ **then**
$\quad \mathsf{ComputeUpdates}_{\beta\,s}(pos, \gamma)$
else
$\quad$ **if** $appliedUpdates(pos)$ **then**
$\quad\quad$ **if** $exitCond(^{\alpha}v)$ **then**
$\quad\quad\quad$ PopState
$\quad\quad\quad [\![pos]\!] := (\mathtt{undef}, composedUpdates(pos), \mathtt{undef})$
$\quad\quad\quad lastUpdates(pos) := composedUpdates(pos)$
$\quad\quad$ **else**
$\quad\quad\quad currUpdates(pos) := \mathtt{undef}$
$\quad\quad\quad \mathsf{PrepareNextStep}(^{\gamma}v)$
$\quad\quad\quad appliedUpdates(pos) := false$
$\quad\quad\quad \mathsf{ClearTree}(\gamma)$
$\quad\quad\quad rulePos(\gamma) := pos$
$\quad\quad\quad pos := \gamma$
$\quad$ **else**
$\quad\quad$ **let** $uSet = \mathsf{Aggregate}(currUpdates(pos))$,
$\quad\quad\quad composed \leftarrow \mathsf{Compose}(composedUpdates(pos),$
$\quad\quad\quad\quad\quad\quad\quad\quad currUpdates(pos))$ **in**
$\quad\quad\quad\quad composedUpdates(pos) := composed$
$\quad\quad\quad\quad lastUpdates(pos) := currUpdates(pos)$
$\quad\quad\quad\quad$ **if** $aggregationConsistent(currUpdates(pos)) \wedge$
$\quad\quad\quad\quad\quad isConsistent(uSet)$ **then**
$\quad\quad\quad\quad\quad\quad \mathsf{Apply}(uSet)$
$\quad\quad\quad\quad\quad\quad appliedUpdates(pos) := true$
$\quad\quad\quad\quad\quad\quad \mathsf{ClearTree}(\alpha)$
$\quad\quad\quad\quad\quad\quad pos := \alpha$
$\quad\quad\quad\quad\quad$ **else**
$\quad\quad\quad\quad\quad\quad [\![pos]\!] := (\mathtt{undef}, composed, \mathtt{undef})$
$\quad\quad\quad\quad\quad\quad currUpdates(pos) := \mathtt{undef}$
$\quad\quad\quad\quad\quad\quad$ PopState

The evaluation of a **do**-block statement begins with the interpretation of the *execution condition*. If the execution condition is met, the rules to be enabled are chosen following the *selection clause*. Then, based on the *execution strategy*, the updates are computed.

We define two function *enterCond* and *exitCond* applied to a pair as, respectively, the projection on the first and second element of it. Since the computation of the updates for a single step of the **do**-block could require several succeeding actions we use the location *composedUpdates* to accumulate and store the intermediate provisional updates which contributes to the final update set of the rule.

Some termination condition (e.g. fixpoint and noupdates) could require a check on the update set produced during the computation of a **do**-block step. For this reason we store the current updates in the location *currUpdates* and the last computed update set in the location *lastUpdates*. The condition on the updates requires the location *rulePos* to be initialised with the executing

rule *pos*. So we forward it to the clauses that could require a termination condition check (α, γ).

When the execution condition and the selection clause have been evaluated, the process continues with the computation of a step that will assign the produced updates to the location *currUpdates*.

The macro ComputeUpdates is parametric with respect to the execution strategy. After the execution of a single step, the **do**-block evaluation proceed aggregating the updates of such step and checking the exit condition. If more steps are to be computed, the PrepareNextStep macro arranges the selected rules initialising the next compute step accordingly with the execution strategy.

Here we define three of the possible strategies that cover the update set computation of basic and Turbo ASMs. We also include the stepwise strategy that in most cases improves the specification of control state ASMs.

The parallel strategy selects non-deterministically the unevaluated rules from the pool of selected rules stored in ${}^{\gamma}v$, and produces as update the union of all the executed rules updates. The semantic of this strategy corresponds to the default parallel behaviour of the ASM rules.

Parallel Strategy

ComputeUpdates$_{inparallel}(pos, \gamma) \equiv$
  choose $\lambda \in {}^{\gamma}v$ **with** $\neg evaluated(\lambda)$ **do**
    $pos := \lambda$
  ifnone
    $currUpdates(pos) := \bigcup_{i \in {}^{\gamma}v} updates(i)$

The stepwise strategy evaluates the list of enabled rules one by one for each step of the **do**-block evaluation. This behaviour is achieved keeping an internal implicit program counter that chooses, step by step, the next rule to be executed.

If a **do**-block statement defines a repeating execution condition (e.g. *while*), an executing machine step corresponds to the aggregation of the sequence of update sets generated by the application of the execution strategy. The list of rules to be executed are stored, like for the other strategies, into ${}^{\gamma}v$.

Stepwise Strategy

ComputeUpdates$_{stepwise}(pos, \gamma) \equiv$
  if $\neg evaluated(head({}^{\gamma}v))$ **then**
    $pos := head({}^{\gamma}v)$
  else
    $currUpdates(pos) := updates(head({}^{\gamma}v))$

The sequence strategy executes all the selected rules one after the other in a provisional state. The composition of these updates is the result of a single step with the sequence strategy. This strategy replaces the **seqblock** rule of TurboASMs. In fact, all the selected rules are executed in sequence as a single step of the **do**-block evaluation. As for the other strategies, also *sequence* may execute the selected rules multiple times depending on the execution condition.

Sequence Strategy

```
ComputeUpdates_sequence(pos, γ) ≡
  if seqUpdates(pos) = undef then
    seqUpdates(pos) := {}
    PushState
  else
    if ^γv = [] then
      PopState
      currUpdates(pos) := seqUpdates(pos)
    else
      if ¬evaluated(head(^γv)) then
        pos := head(^γv)
      else
        let uSet = Aggregate(updates(head(^γv))),
            composed ← Compose(seqUpdates(pos), updates(head(^γv))) in
          seqUpdates(pos) := composed
          if aggregationConsistent(updates(head(^γv)))∧
            isConsistent(uSet) then
              Apply(uSet)
              [[γ]] := (undef, undef, tail(^γv))
          else
              PopState
              currUpdates(pos) := composed
```

The PrepareNextStep macro ensures that, if the *exit condition* is not met, the enabled rules for the next computation step of the **do**-block are in the correct state. For example the node trees of the rules are cleared in order to make them ready for a new evaluation, and (for the *stepwise* strategy) the rule list is modified in order to maintain the internal counter of such rules.

do-block macros

```
PrepareNextStep(strategy, γ) ≡
  case strategy of
    inparallel :
      forall r ∈ ^γv do
        ClearTree(pos(r))
    stepwise :
      if |^γv| ≤ 1 then
        forall r ∈ storedRules(γ) do
          ClearTree(pos(r))
        [[γ]] := (undef, undef, storedRules(γ))
      else
        [[γ]] := (undef, undef, tail(^γv))
    insequence :
      forall r ∈ storedRules(γ) do
        ClearTree(pos(r))
      [[γ]] := (undef, undef, storedRules(γ))
```

In the following sections, the clause definitions are intended to be used only inside a **do**-block statement. The evaluation of any occurrences of them in other contexts is undefined.

The body of the semantic contains some macros that have been defined in [37]. We only give the intuitive semantic for them. The macros PushState and PopState are responsible for saving and restoring the state of the interpreter. The Apply rule applies an update set to the current state.

4.6.1 Execution Conditions

In this section we define some execution condition clauses. An execution condition evaluates to a pair of boolean values. The **do**-block interprets the first element in the pair as enter condition

while the second as exit condition. The set of execution conditions we define here is not exhaustive, any new execution condition could be added similarly. Nevertheless it covers basic and turbo ASMs.

The **if** execution condition allows the **do**-block body to be executed when its condition is true and, after the execution of a single step, the **do**-block rule terminates its computation.

If Execution conditions

$$\begin{array}{lll} (\!|\, \textbf{if}\ {}^{\alpha}\boxed{\mathrm{e}}\ |\!) & \rightarrow & pos := \alpha \\ & & rulePos(\alpha) := rulePos(pos) \end{array}$$

$$(\!|\, \textbf{if}\ {}^{\alpha}v\ |\!) \rightarrow [\![pos]\!] := (\texttt{undef}, \texttt{undef}, \langle {}^{\alpha}v = true, true\rangle)$$

The **while** execution condition allows the **do**-block body to be executed if its condition is true. A new execution step will be performed while the condition holds true.

While Execution conditions

$$\begin{array}{lll} (\!|\, \textbf{while}\ {}^{\alpha}\boxed{\mathrm{e}}\ |\!) & \rightarrow & pos := \alpha \\ & & rulePos(\alpha) := rulePos(pos) \end{array}$$

$$\begin{array}{lll} (\!|\, \textbf{while}\ {}^{\alpha}v\ |\!) & \rightarrow & \textbf{let}\ b = {}^{\alpha}v = \mathsf{true}\ \textbf{in} \\ & & \quad [\![pos]\!] := (\texttt{undef}, \texttt{undef}, \langle b, \neg b\rangle) \end{array}$$

The **until** execution condition always perform at least one step of the **do**-block body. Until the condition becomes true, further steps must be performed.

Until Execution conditions

$$\begin{array}{lll} (\!|\, \textbf{until}\ {}^{\alpha}\boxed{\mathrm{e}}\ |\!) & \rightarrow & pos := \alpha \\ & & rulePos(\alpha) := rulePos(pos) \end{array}$$

$$(\!|\, \textbf{until}\ {}^{\alpha}v\ |\!) \rightarrow [\![pos]\!] := (\texttt{undef}, \texttt{undef}, \langle \mathsf{true}, {}^{\alpha}v = true\rangle)$$

The **times** execution condition executes the **do**-block body a specified number of times.

times

$$\begin{array}{lll} (\!|\, {}^{\alpha}\square\ \textbf{times}\ |\!) & \rightarrow & \textbf{if}\ times(pos) = \texttt{undef} \vee times(pos) \leq 0\ \textbf{then} \\ & & \quad pos := \alpha \\ & & \textbf{else} \\ & & \quad times(pos) := times(pos) - 1 \\ & & \quad [\![pos]\!] := (\texttt{undef}, \texttt{undef}, \langle times(pos) > 0, times(pos) \leq 0\rangle) \end{array}$$

$$\begin{array}{lll} (\!|\, {}^{\alpha}v\ \textbf{times}\ |\!) & \rightarrow & \textbf{if}\ {}^{\alpha}v \in \textsc{Number}\ \textbf{then} \\ & & \quad times(pos) := {}^{\alpha}v - 1 \\ & & \quad [\![pos]\!] := (\texttt{undef}, \texttt{undef}, \langle {}^{\alpha}v > 0, {}^{\alpha}v \leq 0\rangle) \\ & & \textbf{else} \\ & & \quad \mathsf{Error}(\text{“The parameter is not a number”}) \end{array}$$

The **forever** execution condition always permit entering the **do**-block but does not let the execution to exit it. Notice that such condition does not permit the observation of the computed steps of the **do**-block evaluation. If observing the evolution of the machine state is the intended behaviour, an **if** *true* condition preferable.

forever

$$(\!|\, \textbf{forever}\, |\!) \rightarrow [\![pos]\!] := (undef, undef, \langle true, false\rangle)$$

For practical reasons of specification simulation it is possible to define the **always** execution condition as syntactic sugar for “**if** *true*”.

The **once** execution condition performs the **do**-block exactly one time. After the first execution the associated rule will always produce an empty update set.

Once

$$
\begin{array}{lll}
(\!|\,\textbf{once}\,|\!) \rightarrow & \textbf{if } executed(rulePos(pos)) = \texttt{undef} \textbf{ then} \\
& \quad executed(rulePos(pos)) = true \\
& \quad [\![pos]\!] := (\texttt{undef}, \texttt{undef}, \langle true, true \rangle) \\
& \textbf{else} \\
& \quad [\![pos]\!] := (\texttt{undef}, \texttt{undef}, \langle false, true \rangle)
\end{array}
$$

The following definitions add two boolean expressions to be used as termination conditions: **no updates** , **fixpoint** . The **no updates** expression evaluates to true if the associated **do**-block have produced an empty update set. The **fixpoint** expression checks if the associated **do**-block have produced an update set that does not alter the state.

No updates and fixpoint expression

$$
\begin{array}{l}
(\!|\,\textbf{no updates}\,|\!) \rightarrow \\
\quad \textbf{let } u = lastUpdates(rulePos(pos)) \textbf{ in} \\
\quad\quad [\![pos]\!] := (\texttt{undef}, \texttt{undef}, u = \texttt{undef} \vee u = \{\}) \\
\\
(\!|\,\textbf{fixpoint}\,|\!) \rightarrow \\
\quad \textbf{if } lastUpdates(rulePos(pos)) \neq \texttt{undef} \wedge currUpdates(rulePos(pos)) \neq \texttt{undef} \textbf{ then} \\
\quad\quad \textbf{let } isFixPoint = \forall \langle l, u, v \rangle \in currUpdates(rulePos(pos)). \\
\quad\quad\quad\quad\quad\quad \langle l, u, v' \rangle \in lastUpdates(rulePos(pos)) \Rightarrow v = v' \textbf{ in} \\
\quad\quad\quad [\![pos]\!] := (\texttt{undef}, \texttt{undef}, isFixPoint) \\
\quad \textbf{else} \\
\quad\quad [\![pos]\!] := (\texttt{undef}, \texttt{undef}, false)
\end{array}
$$

In order to access the information about the state evolution of a **do**-block evaluation, we assume that the *rulePos*, and *lastUpdates* locations are initialised. The first represents a reference to the parent **do**-block node for which the condition will check the updates state. The second contains the last computed updates set. For this reason in Section 4.6 the semantic rules set them.

4.6.2 Selection Clause

The selection clause is responsible for choosing rules that concur to the update set production of a **do**-block rule. We foresee three kind of selection clauses that cover most of the common usage cases: one, all, any. If more selection clauses are needed, the semantic can be extended in the same flavor of the following definitions. As suggested by their names, the **one of** , **any of** and **all of** selection clauses choose, respectively, one, a subset and all the rules in the **do**-block body. The selection is fixed the first time each clause is evaluated. The selected rules are stored in the location *storedRules*. If the selection process should be performed again, we provide the selection clause with an optional **reset on** condition. When the **reset on** condition is true, a new selection is performed and the new list of selected rules is stored in *storedRules*. We have chosen to store the selected rules as a list since some execution strategy may depend on the textual order of such rules (e.g. **stepwise**).

All of rule

$$
\begin{array}{l}
(\!|\,\textbf{all of } {}^{\lambda_1}[?]^{\lambda_2}[?] \ldots {}^{\lambda_n}[?]\,|\!) \rightarrow \\
\quad \textbf{let } selectedRules = [\mathsf{CopyTree}(\lambda_1), \ldots, \mathsf{CopyTree}(\lambda_n)] \textbf{ in} \\
\quad\quad [\![pos]\!] := (\texttt{undef}, \texttt{undef}, selectedRules) \\
\quad\quad storedRules(pos) := selectedRules
\end{array}
$$

For the **all of** selection clause, the **reset on** condition is trivial since reselection computes always the same rules set. For this reason the semantic does not provide a definition of **reset on** for this case.

One of rule

$$
\begin{array}{l}
(\!|\textbf{one of } {}^{\lambda_1}[\,?\,]{}^{\lambda_2}[\,?\,]\ldots{}^{\lambda_n}[\,?\,]|\!) \rightarrow \quad \textbf{if } \mathit{storedRules}(\mathit{pos}) = \texttt{undef} \textbf{ then} \\
\qquad\qquad\qquad\qquad\quad \textbf{choose } \lambda_i \in [\lambda_1, \ldots, \lambda_n] \textbf{ do} \\
\qquad\qquad\qquad\qquad\qquad \textbf{let } \mathit{selectedRules} = [\mathsf{CopyTree}(\lambda_i)] \textbf{ in} \\
\qquad\qquad\qquad\qquad\qquad\quad [\![\mathit{pos}]\!] := (\texttt{undef}, \texttt{undef}, \mathit{selectedRules}) \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{storedRules}(\mathit{pos}) := \mathit{selectedRules} \\
\qquad\qquad\qquad\qquad \textbf{else} \\
\qquad\qquad\qquad\qquad\quad [\![\mathit{pos}]\!] := (\texttt{undef}, \texttt{undef}, \mathit{storedRules}(\mathit{pos}))
\end{array}
$$

$$
\begin{array}{l}
(\!|\textbf{one of } {}^{\lambda_1}[\,?\,]{}^{\lambda_2}[\,?\,]\ldots{}^{\lambda_n}[\,?\,] \textbf{ reset on } {}^{\alpha}[\,e\,]|\!) \rightarrow \quad \textbf{if } \mathit{storedRules}(\mathit{pos}) \neq \texttt{undef} \textbf{ then} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{pos} := \alpha \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{choose } \lambda_i \in [\lambda_1, \ldots, \lambda_n] \textbf{ do} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{let } \mathit{selectedRules} = [\mathsf{CopyTree}(\lambda_i)] \textbf{ in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad [\![\mathit{pos}]\!] := (\texttt{undef}, \texttt{undef}, \mathit{selectedRules}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{storedRules}(\mathit{pos}) := \mathit{selectedRules}
\end{array}
$$

$$
\begin{array}{l}
(\!|\textbf{one of } {}^{\lambda_1}[\,?\,]{}^{\lambda_2}[\,?\,]\ldots{}^{\lambda_n}[\,?\,] \textbf{ reset on } {}^{\alpha}v|\!) \rightarrow \quad \textbf{if } {}^{\alpha}v = \mathsf{true_e} \textbf{ then} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{choose } \lambda_i \in [\lambda_1, \ldots, \lambda_n] \textbf{ do} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{let } \mathit{selectedRules} = [\mathsf{CopyTree}(\lambda_i)] \textbf{ in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad [\![\mathit{pos}]\!] := (\texttt{undef}, \texttt{undef}, \mathit{selectedRules}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{storedRules}(\mathit{pos}) := \mathit{selectedRules} \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad [\![\mathit{pos}]\!] := (\mathit{undef}, \mathit{undef}, \mathit{storedRules}(\mathit{pos}))
\end{array}
$$

The **one of** clause chooses one of the rules and assigns it to the *storedRules* location that contains a copy of the selected rule in case of multiple steps in the evaluation of a **do**-block rule. When specified, the **reset on** clause is evaluated if the *storedRules* location is defined, otherwise it means that it is the first evaluation and instead of checking the reset condition, one of the rules must be selected. If the reset condition is evaluated and holds the *true* value, a new selection is performed updating the *storedRules*. Otherwise the already selected rules are returned as value.

Any of rule

$$
\begin{array}{l}
(\!|\textbf{any of } {}^{\lambda_1}[\,?\,]{}^{\lambda_2}[\,?\,]\ldots{}^{\lambda_n}[\,?\,]|\!) \rightarrow \\
\qquad \textbf{choose } \{\lambda_{n_1}, \ldots, \lambda_{n_k}\} \sqsubseteq^* \{\lambda_1, \ldots, \lambda_n\} \textbf{ do} \\
\qquad\quad \textbf{let } \mathit{selectedRules} = [\mathsf{CopyTree}(\lambda_{n_1}), \ldots, \mathsf{CopyTree}(\lambda_{n_k})] \textbf{ in} \\
\qquad\qquad [\![\mathit{pos}]\!] := (\texttt{undef}, \texttt{undef}, \mathit{selectedRules}) \\
\qquad\qquad \mathit{storedRules}(\mathit{pos}) := \mathit{selectedRules}
\end{array}
$$

$$
\begin{array}{l}
(\!|\textbf{any of } {}^{\lambda_1}[\,?\,]{}^{\lambda_2}[\,?\,]\ldots{}^{\lambda_n}[\,?\,] \textbf{ reset on } {}^{\alpha}[\,e\,]|\!) \rightarrow \\
\qquad \textbf{if } \mathit{storedRules}(\mathit{pos}) \neq \texttt{undef} \textbf{ then} \\
\qquad\quad \mathit{pos} := \alpha \\
\qquad \textbf{else} \\
\qquad\quad \textbf{choose } \{\lambda_{n_1}, \ldots, \lambda_{n_k}\} \sqsubseteq^* \{\lambda_1, \ldots, \lambda_n\} \textbf{ do} \\
\qquad\qquad \textbf{let } \mathit{selectedRules} = [\mathsf{CopyTree}(\lambda_{n_1}), \ldots, \mathsf{CopyTree}(\lambda_{n_k})] \textbf{ in} \\
\qquad\qquad\quad [\![\mathit{pos}]\!] := (\texttt{undef}, \texttt{undef}, \mathit{selectedRules}) \\
\qquad\qquad\quad \mathit{storedRules}(\mathit{pos}) := \mathit{selectedRules}
\end{array}
$$

$$
\begin{array}{l}
(\!|\textbf{any of } {}^{\lambda_1}[\,?\,]{}^{\lambda_2}[\,?\,]\ldots{}^{\lambda_n}[\,?\,] \textbf{ reset on } {}^{\alpha}v|\!) \rightarrow \\
\qquad \textbf{if } {}^{\alpha}v = \mathsf{true_e} \textbf{ then} \\
\qquad\quad \textbf{choose } \{\lambda_{n_1}, \ldots, \lambda_{n_k}\} \sqsubseteq^* \{\lambda_1, \ldots, \lambda_n\} \textbf{ do} \\
\qquad\qquad \textbf{let } \mathit{selectedRules} = [\mathsf{CopyTree}(\lambda_{n_1}), \ldots, \mathsf{CopyTree}(\lambda_{n_k})] \textbf{ in} \\
\qquad\qquad\quad [\![\mathit{pos}]\!] := (\texttt{undef}, \texttt{undef}, \mathit{selectedRules}) \\
\qquad\qquad\quad \mathit{storedRules}(\mathit{pos}) := \mathit{selectedRules} \\
\qquad \textbf{else} \\
\qquad\quad [\![\mathit{pos}]\!] := (\mathit{undef}, \mathit{undef}, \mathit{storedRules}(\mathit{pos}))
\end{array}
$$

The semantic of **any of** selection clause is similar to the **one of** but here the selected rules are a subset of all the available rules. With $\sqsubseteq^*$ we denote the sub-sequence operator that, given a list, returns a sub-list such that $n_i \in \{1, \ldots, n\}$ and $i < j \Rightarrow n_i < n_j$.

4.7 Extending the do-block construct

The **do**-block we presented in this chapter covers most of the common control flow construct that have appeared in the history of ASMs. The semantic has been presented split into different clauses not only to make it more readable but also to prepare it for future extensions.

The construct can be extended in each of its components: execution condition, strategy and rule selection. We foresee possible extensions since in the past several addition to the ASM language have been made, so we wanted to make the construct general enough to not invalidate it for future needs.

To extend the **do**-block with a new execution condition one must provide a semantic that evaluates to a pair of boolean. It is possible to think also at the simple case in which the pair itself represents the execution condition. If the elements of the pair are bounded to location values, the execution of the do block could be instrumented by the logic defined inside its body. One disadvantage of this kind of execution condition, and the main reason we have not listed it among the other execution conditions, it that the behaviour of the rule may become easily obscure and as a consequence abdicate to the readability goal of the construct.

Extending the strategies requires to provide a new ComputeUpdates macro for the intended behaviour that sets *currUpdates* location when its computation is complete. Also PrepareNextStep macro that is aware of the new strategy must be extended. Some fancy example of other execution strategies may be:

- lexical reversed order
- a generalised stepwise that executes subsets of the enabled rules step by step instead of just one
- asynchronous execution of a subset of the rules (managing each rule as the execution of a separate agent with the rule as *program*)

Finally the extension of the rule selection clause only requires to provide the semantic for the selection that sets the *storedRules* location. An example of another rule selection clause may be indexed selection. In other words the ability to choose rules accessing them by properties of their lexical position.

Chapter 5

Modularity as Services

In this chapter we address modularity from the point of view of structuring a specification into independent units of behaviour (the modules). We describe how modular decomposition improves the specification process and then evaluate the current status with regard to module support in the ASM language. The chapter concludes with the description of our model proposal.

5.1 Modular decomposition

One of the most valuable technique in software engineering is *decomposition*. The principle of deconstructing a problem into smaller and more manageable parts has been addressed in several ways and at different levels of abstraction.The decomposition produces different sub-component often called modules, each of them:

- represents a smaller and possibly easier problem to address;
- may be solved independently from the others;
- shifts the main problem concerns to the communication among such modules.

5.1.1 Why decomposing?

The first motivation for the decomposition of a system into modules is related to its complexity. When dealing with large systems, it is not feasible to maintain all the information about its functionalities in a single place both because it is an hardly manageable task and because it is difficult to track all the dependencies between them. Identifying sets of related features and categorising them instead reduces the effort needed to understand the behaviour of the system. Most programming languages provide some sort of modular constructs. For example, object oriented languages group unit of behaviour into classes that represent an abstraction of all the possible entities sharing the same capabilities. Similarly, specification languages are endowed with modularization mechanisms. The main reason of having such mechanisms is that they improve the readability and slice the problem into different abstract components that do not require a full understanding of all the detail they cover providing an abstract view of their sub-domains.

Modular decomposition is also useful for information hiding. Usually modules provide an interface through which it is possible to interact with its functionalities but all the internal logic is not accessible or at least not visible from the user point of view. This structure allows to separate the sub-component logic from the way to access it.

Decomposing with modules create a twofold abstraction mechanism. On the horizontal level, all the modules that communicates among each other give a perspective at the same abstraction level. On a vertical level looking inside a module allow system details to emerge.

Modular decomposition improves the modelling process reducing a big problem into smaller problems. Moreover each sub-problem tends to be orthogonal to the others describing the initial

system as a set of independent entities with their own responsibilities. From a specification point of view, this means that is possible to divide the task of producing the specification for a large system among various people. Each of them can focus on a specific aspect of the initial problem without worrying about the other details.

Finally, a module can also be seen as a behaviour container. This approach allows to create specification as composition of instances of already available modules and to decompose the specification into sub-modules that address sub-problems.

5.1.2 Current support for modules of the ASM language

In this section we analyse the current module support for the ASM language. In the ASM method the main elements of decomposition are rules, TurboASMs, agents and standard syntactical module[12].

Rules can be used at every desired granularity. A rule can specify a single action or a whole process via refinement steps. They represent a form of parametric macro definition. Most of the ASM models of complex systems (e.g. the Java Virtual Machine) are defined as parallel execution of macros. The downside of using macro as modules is that macros are not independent. When put together in parallel each macro must be checked to not interfere with the others in order to avoid inconsistent updates. This process is error prone since it requires to carefully look inside all the other macros for inconsistencies. The lack of independence make rules not suitable to be used as modules.

Turbo ASMs instead encapsulate the behaviour but not the structure of a system. From a functional point of view, Turbo ASMs structure the computation as a whole with return value while modules are often required to handle different actions. For example classes are containers with a set of methods that implement as functions their behavioural features.

An agent is a independent ASM running its program. The communication between agents is delegated to synchronisation with other agents using some state location that are read and modified accordingly to the communication logic. Agents appear to be a good candidate for the definition of a module. Modules defined as agents have the characteristic to represent active unit of behaviour and not only a collection of static behaviour definitions. Moreover, decomposing by agents may also shows the structural view of a system. Yet the independence of agents in the ASM language is not complete since it is achieved by convention, parameterising all the locations inside the program of an agent by the *self* special function that evaluates to the unique reference to the running agent. The self function has also the goal to allow the execution of different agent with the same program.

The ASM book [12] also defines a syntactical module to structure large ASMs (see Chapter 2). Such module definition is a collection of static rules and function definitions and assumes no name clashes. To the best of our knowledge they have never been used, since their definition is not completely clear and their practical use is inconvenient.

The proposal of Ernst et al. [36] introduces the concept of operation call from a machine to another and replacement via modular refinement (see Section 2.3.6 for further details). The only problem is that it is defined on a variant of ASM and not on the classic definition. We would like to remain as close as possible to the standard semantic. Moreover the definitions for modular decomposition require to be careful in the creation of machines and submachines, in particular regarding the constraint about avoiding using internal functions of a submachine outside the set of operations. Each operation requires that some preconditions are met and a set of input and output location parameterizes the operation and stores the results. A similar idea, but tailored for hardware logic component specification, has been proposed by Shimd [81], where each component contains input and output lines.

It seems that there is no common established notion of module but it appears that the module should provide a set of accessible operation, possibly hiding the internal logic, and that can be considered as a running machine (or agent) or as a static collection of rules. The current module definitions require to meet some construction criteria when modelling yet it would be more practical, from a pragmatic point of view, to give more freedom about function and location names as well as operation call behaviour.

5.2 Towards a module definition

From the perspective that emerges from the current status of ASM support for modules and from the various module and component definitions that have appeared as a result of the ASM method application during the years, a module should encompass the boundaries of a sub-problem. A module may be a static or active entity since it can be seen as a process whose behaviour implements the desired sub-component or as a set of aggregated definitions that can be performed when needed. The definition of a module should be abstract enough to allow the specification of any kind of component from web services with their REST APIs [38] to protocols. With regard to protocol definition, usually during the specification of a process (for example business processes) various components must interact with a specific interface but the definition of the communication protocol is not important. An example of this situation is [93] where in order to define a business process, also a model for the communication protocol has been provided. So we would like to endow modules with the ability to abstract from the actual protocol definition when not needed.

In this chapter we will provide a definition of such modules. We believe that our definition provides an improvement over the current state of the modelling process with ASMs. In particular, a large set of models will be naturally expressed by our module definition, for example services orchestration, business processes and in general sub-systems description.

The ASM agents are already a good behavioural description of an independent entity that runs its program. So the behavioural aspect of modules can be captured by the definition of agents. Unfortunately, agents do not provide clear interface definition that establish a contract for the synchronisation with other agents. Moreover, the communication between agents is not standardised: several versions of the definition of ASM runs have been proposed depending on agent synchronisation. To address these two aspects we should enrich the agent with a communication abstraction for requiring the module to perform certain behaviours and to let the module be aware of such request to perform them.

The availability of module definitions will also be reflected on the models definition process. A model may become naturally a composition of, potentially already defined, modules. This means that modules also improve the reuse of specifications, and allow different people to work in parallel on different parts of the model. In order to provide such flexibility, modules should be retrievable so one can fetch the modules needed to compose a new one as the current project management systems actually do for programming languages. Platforms like Maven [75] implements a mechanism for software component organisation to handle components dependencies. We would like to pave the way for a similar system but in the world of specifications through module retrieval.

In the following sections we will introduce the syntax and semantic for our proposal of ASM modules. The semantic is given in terms of extension of the CoreASM interpreter and has been inspired by ADA's tasks with rendezvous[28].

Our module definition will allow to define:

- a behaviour
- a compact representation of interface reflecting its capabilities
- an abstraction of the communication protocol to interact with the module itself
- a clear process to instantiate and access it

We propose modules as values of the MODULE domain. A module is an active unit of behaviour that is able to accept and fulfil requests based on its internal state. Module instances extend the *Agent* set with a new agent whose program describes the module behaviour. The communication with a module instance is performed with two new transition rules *request* and *accept* that together abstract the communication protocol between modules. Finally modules can be imported with the new *import* transition rule that is in charge of retrieving a module instance.

5.3 Modular Services

We define an ASM module as an agent together with a set of offered services. With the word *service* we mean every functionality or process that the specifier needs to describe the behaviour of the module and that wants to expose to other modules. Each service can be requested to the module that will eventually perform it according to its behaviour. Services will eventually produce results to the requester.

The agent program determines the module internal logic. A service offered by a module is defined by an *entry* $e \in$Entry that is identified by its name, a list of parameters and a list of output locations. The list of parameters represents the required data that are needed to perform the service, the output parameters, instead, represent the expected locations in which the relevant portion of the computed module state is stored to be accessible to the service client. Together name, input and output parameters establish the contract between the module and its clients for a single service. The set of entries of a module can be seen as the interface to the it. For each module we assume that the following functions are defined, otherwise the interaction with the module is not possible:

$$\begin{aligned} name_m &: \textsc{Module} \to \textsc{Name} \\ entries_m &: \textsc{Module} \to \textsc{Set}(\textsc{Entry}) \\ agent_m &: \textsc{Module} \to \textsc{Agent} \\ requests_m &: \textsc{Module} \to \textsc{Set}(\textsc{Request}) \\ module_m &: \textsc{Name} \to \textsc{Module} \end{aligned}$$

Where given a module, $name_m$ is the name associated with it, $entries_m$ holds the set of entries exposed by the module, and $agent_m$ is the agent associated with the module. The function $requests_m$ contains, for each module, a set of pending requests for services whose elements are in the domain Request. The $requests_m$ of a module behaves like a mailbox where requests arrive. The management of this mailbox depends on the module logic.

Regarding the elements in the Entry domain we assume that the following functions are defined:

$$\begin{aligned} name_e &: \textsc{Entry} \to \textsc{Name} \\ inParams_e &: \textsc{Entry} \to \textsc{List}(\textsc{Name}) \\ outParams_e &: \textsc{Entry} \to \textsc{List}(\textsc{Name}) \\ module_e &: \textsc{Entry} \to \textsc{Module} \\ getEntry &: \textsc{Request} \to \textsc{Entry} \\ entryByName_e &: \textsc{Name} \to \textsc{Set}(\textsc{Entry}) \end{aligned}$$

Similarly to modules, $name_e$ holds the name of the given entry element. The function $module_e$ refers to the module the entry belongs to, and $inParams_e$, $outParams_e$ return respectively the input and output parameters for the entry. It is possible to retrieve the name of an entry from a request using the function $getEntry$. The function $entryByName_e$ retrieves all the entry elements with a given name. Notice that the same name can be used in different modules.

Finally we introduce some functions on the Request domain to manage service requests. The Request domain contains the service requests and the following functions define the data associated to them.

$$entry_r : \textsc{Request} \to \textsc{Entry}$$
$$params_r : \textsc{Request} \to \textsc{List}(\textsc{Element})$$
$$result_r : \textsc{Request} \to \textsc{List}(\textsc{Element})$$

The function $entry_r$ gets the entry for which the request has been created. The request data parameters are stored by $params_r$ and the result of the request after its completion is accessible from $result_r$.

In order to perform services, a module first must be imported. Importing a module creates an instance of a module (an element of MODULE with the related functions initialised). We propose two forms of module import: an *explicit* module expression and an *implicit* module creation from a source. In the following sections we introduce these two constructs and the interaction mechanism to request and perform module services.

5.3.1 Creating Modules

In this section we introduce two expression, **module** and **fetch** to declare in a compact way a module and to get a module from a given source.

Definition 5.3.1. (Module declaration) A module declaration is an expression:

Module declaration

module m **with**
  entry $e_1(x_1^1, \ldots, x_{i_1}^1)$ **into** $y_1^1, \ldots, y_{j_1}^1$
  entry $e_2(x_1^2, \ldots, x_{i_2}^2)$ **into** $y_1^2, \ldots, y_{j_2}^2$
  ...
  entry $e_n(x_1^n, \ldots, x_{i_n}^n)$ **into** $y_1^n, \ldots, y_{j_n}^n$
  do
    R

where m is the module name, e_i are the entries of the module and R is the body, a rule defining the module behaviour.

A module declaration extends the AGENT domain with a new agent a whose program is R and the MODULE domain with a new element e with:

- $name_m(e) = m$
- $agent_m(e) = a$
- $module(a) = m$
- $requests_m(e) = \{\}$
- $module_m(m) = e$
- $entries_m(e) = \{d_1, \ldots, d_n\}$

For each element $d_h \in entries_m(e)$, $d_h \in$ ENTRY and the entry related function are initialized with:

- $name_e(d_h) = e_h$
- $inParams_e(d_h) = [x_1^i, \ldots, x_{i_h}^i]$
- $outParams_e(d_h) = [y_1^i, \ldots, y_{j_h}^i]$
- $d_h \in entryByName_e(e_h)$

Each entry is defined by its name e_h and two sets of identifiers: the input parameters $x^h_1, \ldots, x^h_{i_h}$ and the output locations $y^h_1, \ldots, y^h_{j_h}$. Entries only define the interface to access module services, the actual behaviour is defined inside the module body as we describe in Section 5.3.3.

Instead of defining the entries set, we could avoid writing it and just rely on the **accept** rule that is in charge of performing a service inside the module body. We have chosen to keep entry definitions as they constitute a compact view of the module capabilities without the need of reading also its behaviour. In this way the readability and understandability of the module improve.

The formal semantics for an interpreter of this construct is the following:

Module creation

$(\!|$ **module** $^{\alpha}x$ **with** $^{\lambda_1}\boxed{\text{e}} \ldots {}^{\lambda_n}\boxed{\text{e}}$ **do** $^{\beta}\boxed{\Gamma}$ $|\!) \rightarrow$
  choose $\lambda \in \{\lambda_1, \ldots, \lambda_n\}$ **with** $\neg evaluated(\lambda)$ **do**
    $pos := \lambda$

$(\!|$ **module** $^{\alpha}x$ **with** $^{\lambda_1}v \ldots {}^{\lambda_n}v$ **do** $^{\beta}\boxed{\Gamma}$ $|\!) \rightarrow$
  if $\exists \lambda \in \{\lambda_1, \ldots, \lambda_n\} \wedge {}^{\lambda}v \notin$ MODULE **then**
    Error("Non entry value in module definition")
  else
    extend MODULE **with** m **do**
      $name_m(m) := {}^{\alpha}x$
      forall $e \in \{{}^{\lambda_1}v, \ldots, {}^{\lambda_n}v\}$ **do**
        $module_e(e) := m$
      $entries_m(m) := \{{}^{\lambda_1}v, \ldots, {}^{\lambda_n}v\}$
      extend AGENTS **with** a **do**
        $agent_m(m) := a$
        $module(a) := m$
        let $mainRule =$ CopyTree$(body(ruleValue(\beta)), true)$ **in**
          $program(a) := mainRule$
          SetParentModule$(mainRule, m)$
      $[\![pos]\!] := (\texttt{undef}, \texttt{undef}, m)$

Module creation is composed mainly of entry and body definitions. The body definition ($^{\beta}\boxed{\Gamma}$) is a rule that describes the module behaviour. Entry definitions (λ_i) instead are expressions that evaluates to elements in the ENTRY domain. The identifier $^{\alpha}x$ represents the name of the module. After the evaluation of the entry expressions, if all of them are actually entry elements, the module creation extends MODULE (the universe of modules) with a new one and initialises the functions for it. In particular the entry set ($entries_m$) and the agent that will run the module. Each entry has the referring $module_e$ set to the new module. Finally the new module element is returned as value of the entire expression.

The CopyTree rule shares the definition given in [37] and copies the body of the rule passed as parameter. In order to make the rule call process aware of the modules we provide a refinement of the CopyTreeSub rule; the refinement adds the assignment $module(n) := module(\alpha)$ that allows the **accept** rule to check if the requested entry is in the module contract (see Section 5.3.3). The refinement for CopyTreeSub is defined as follows:

CopyTreeSub refinement

```
CopyTreeSub(α, ⟨x_1, ..., x_n⟩, ⟨λ_1, ..., λ_n⟩) ≡
  if α ≠ undef then
    if class(α) = Id ∧ ∃s.t.token(α) = x_i then
      result ← CopyTree(λ_i, false)
    else
      let n = new(NODE) in
        first(n) ← CopyTreeSub(first(α), ⟨x_1, ..., x_n⟩, ⟨λ_1, ..., λ_n⟩)
        next(n) ← CopyTreeSub(next(α), ⟨x_1, ..., x_n⟩, ⟨λ_1, ..., λ_n⟩)
        class(n) := class(α)
        pattern(n) := pattern(α)
        token(n) := token(α)
        module(n) := module(α)
        grammarRule(n) := grammarRule(α)
        plugin(n) := plugin(α)
        result := n
  else
    result := undef
```

The rule SetParentModule annotates the rule body with the module reference. The annotation process walks recursively through the AST of the rule passed as parameter and whenever a pos with **accept** pattern is found, the function *module* for that pos is defined with the module value m. This process allows the **accept** statement to check its entry against the module's allowed set. The definition of SetParentModule is the following:

SetParentModule definition

```
SetParentModule(pos, module) ≡
  if pattern(pos) = Accept then
    module(pos) := module
  SetParentModule(first(pos), module)
  SetParentModule(next(pos), module)
```

Each $^{\lambda_i}[\,e\,]$ expression in the module creation construct is checked to be an element of the ENTRY domain. An **entry** expression evaluates to elements of ENTRY and are used to populate the set of offered services of a module. Entries are defined by their name, input and output parameters identifiers. The association between entry and module is made inside the module creation expression assigning $module_m$[1].

Entry creation

```
(| entry ^α x(^λ1 x_1, ..., ^λn x_n) into ^γ1 x, ..., ^γm x |) →
        extend ENTRY with e do
          name_e(e) := ^α x
          inParams_e(e) := ⟨^λ1 x, ..., ^λn x⟩
          outParams_e(e) := ⟨^γ1 x, ..., ^γm x⟩
          [[pos]] := (undef, undef, e)
```

Definition 5.3.2. (Module fetching) The expression **fetch from** s extends the domain MODULE with a new element e that is the module definition retrieved from the source s. The module element has the related functions initializes as shown by the module declaration expression.

Analogously to **module** , the **fetch from** expression evaluates to an element of the MODULE domain. In this case the process of creating and defining a new module element is implicit. This form of expression is meant to retrieve a module from a source given as parameter. Examples of

[1] An extension of this approach would allow an entry to be part of more than one module. In this case instead of assigning $module_m$ to an entry value, the location value will become a set of module elements requiring the insertion of the new module reference in this set.

sources are files, URLs and binaries. The nature of such source is not constrained. We want to give maximal freedom on sources in order to let every potential module provider to be included. The idea is to be able to get interoperability with every that can be described by a module with the assumption that a translation function from such entity to a module element exists. We call this function $fetchModule$ with the following signature:

$$fetchModule : \textsc{Source} \rightarrow \textsc{Module}$$

The function $fetchModule$ is in charge of recognising the source and get a module element from it.

Module fetching

```
(| fetch from ^α□ |) →
        if ¬evaluated(α) then
          pos := α
        else
          let m = fetchModule(^α v) in
            if m ∈ MODULE then
              [[pos]] := (undef, undef, m)
            else
              Error("Unable to import module from source")
```

The simplest example of **fetch from** statement usage consists on considering file paths as resource parameter. The $fetchModule$ function would be implemented in order to look for an ASM specification file and to parse it. If the file contains a module definition, the interpretation of the parsed tree would result in a module element.

5.3.2 Importing Modules

Importing a module permits to implicitly retrieve a module element to request its services. In our setting this operation means simply to assign a module element to a location. There is no need to add other mechanisms. For example, assuming that there exists a repository of modules at *"http://asmmodulesrepo.net"* we could import a module that expose mathematical functions doing simply:

Importing module example

```
mathModule := fetch from "http://asmmodulesrepo.net/math"
```

Assigning $mathModule$ to other locations provide an aliasing mechanism for the same module. If more than one module instance of a kind are needed importing the same module multiple times is enough. The following example shows the difference between aliasing and multiple instantiation for modules.

Aliasing and multiple instantiation example

```
parser := fetch from "http://asmmodulerepo.new/parser"
...
aliasedParser := parser
anotherParser := fetch from "http://asmmodulerepo.new/parser"
```

This approach solve the namespace problem. Each module is evaluated to a module element, so the same module name can be used for different module definitions and the same module definition can be assigned as different module elements to multiple locations.

5.3.3 Requesting and performing a service

The basic interaction with a module is achieved by requesting a service. The outcome of requesting a service is to put a new request inside the module request set and waiting for the answer. A module runs the program of its agent and accordingly to the program definition may accept service requests. In order to enable ASMs for these two operations we extend the set of transition rules by adding **accept** and **request** statements.

$$\textbf{request } m \textbf{ to } e(x_1, \ldots, x_n) \textbf{ into } y_1, \ldots, y_m$$
$$\textbf{accept } e(x_1, \ldots, x_n) \ [\ \textbf{with } e'] \textbf{ do } R$$

The intuitive meaning of the **request** rule is to ask a module m to perform the service e with input parameters the value of $x_1, \ldots, x_n$ terms and to assign the results of the service to the locations $y_1, \ldots, y_m$. Notice that a service request does not require to specify every (input or output) parameter. This means that a module service can have optional parameters (that are assumed to be `undef`) and that the service requester can provide a smaller set of output location with the result of ignoring any other result the service has provided. For example if a module offer the division service:

$$\textbf{entry } \mathit{divide}(x, y) \textbf{ into } \mathit{quotient}, \mathit{remainder}$$

when requesting this service, only the quotient can be of interest. In this case the requester will ignore the remainder specifying only one location for the output parameters:

$$\textbf{request } m \textbf{ to } \mathit{divide}(42, 5) \textbf{ into } \mathit{res}$$

The **accept** rule checks if the service e has been requested; if a request is available it performs the rule R, otherwise the agent continue with its program so the **accept** rule is not blocking.

Each module manages requests using a set as pool of pending requests. Every time a **request** transition rule is triggered, a new request is added to the module pool. The module will eventually consume the requests whenever a corresponding **accept** is triggered. The request-accept mechanism has been inspired by the concept of *tasks* and *rendezvous* of the Ada programming language [28].

Synchronous Service Request

The semantic we define in this section describes a synchronous request-accept communication that is the default behaviour of module communication. We introduce the semantic of a request by providing an extension to the classic ASM calculus and an interpreter definition for it.

$$\frac{\begin{array}{c}[\![t]\!]^S_\zeta \in \textsc{Module} \quad e \in \mathit{entries}_e([\![t]\!]^S_\zeta) \quad \mathit{name}_e(e) = x \\ r \in \mathit{requests}_r([\![t]\!]^S_\zeta)\ \mathit{entry}_r(r) = e\ \mathit{params}_r(r) = \langle [\![t_1]\!]^S_\zeta, \ldots, [\![t_n]\!]^S_\zeta \rangle \\ [\![\textbf{accept } x(y_1, \ldots, y_n) \textbf{ with } \varphi]\!]^{S'}_\zeta = U\ \langle (l_1, v_1), \ldots, (l_m, v_m) \rangle = U\ \restriction_{\mathit{outParams}_e(e)}\end{array}}{[\![\textbf{request } t \textbf{ to } x(t_1, \ldots, t_n) \textbf{ into } x_1, \ldots, x_m]\!]^S_\zeta = \{(x_1, v_1), \ldots, (x_m, v_m)\}}$$

The interpreter semantic definition is the following.

Requesting a service rule

(| **request** $^{\alpha}\square$ **to** $^{\beta}x(^{\lambda_1}\boxed{e}, \ldots, ^{\lambda_n}\boxed{e})$ **into** $^{\gamma_1}\boxed{l}, \ldots, ^{\gamma_m}\boxed{l}$ |) $\rightarrow$
  $pos := \alpha$

(| **request** $^{\alpha}v$ **to** $^{\beta}x(^{\lambda_1}\boxed{e}, \ldots, ^{\lambda_n}\boxed{e})$ **into** $^{\gamma_1}\boxed{l}, \ldots, ^{\gamma_m}\boxed{l}$ |) $\rightarrow$
  if $v \notin$ MODULE **then**
    Error("Requesting to non module")
  else
    choose e **in** $entries_m(^{\alpha}v)$ **with** $name_e(e) = ^{\beta}x$ **do**
      choose $\lambda \in \{\lambda_1, \ldots, \lambda_n, \gamma_1, \ldots, \gamma_m\}$ **with** $\neg evaluated(\lambda)$ **do**
        $pos := \lambda$
      ifnone
        if $\neg requestSent(pos)$ **then**
          let $r = new(\text{REQUEST})$ **in**
            $entry_r(r) := e$
            $params_r(r) := \langle \lambda_1, \ldots, \lambda_n \rangle$
            $request(pos) := r$
            $requestSent(pos) := true$
            add r **to** $requests_m(^{\alpha}v)$
        else
          if $requestFulfilled(pos)$ **then**
            let $\langle v'_1, \ldots, v'_{m'} \rangle = result_r(request(pos))$ **in**
            let $k = min(m', m)$ **in**
              $[\![pos]\!] := (\texttt{undef}, \{\!|\langle ^{\gamma_1}l, v'_1\rangle, \ldots, \langle ^{\gamma_k}l, v'_k\rangle, \langle ^{\gamma_{k+1}}l, \texttt{undef}\rangle, \ldots, \langle ^{\gamma_m}l, \texttt{undef}\rangle|\!\}, \texttt{undef})$
              $requestSent(pos) := false$
    ifnone
      Error("Entry not found in module")

The first step for requesting a service is to check if the element we are asking to perform it ($^{\alpha}\boxed{?}$) is indeed a module element. Every expression that evaluates to an element in the MODULE domain is allowed in the α position. If it is a module, the evaluation proceeds looking for an entry among the module $entries_m$, if any, with the same requested name $^{\beta}x$. At this point a new request element r is inserted into the $requests_m$ pool of the module waiting for the service to be performed. When the request is fulfilled, the location designated to hold the service output are assigned producing the resulting **request** update set. The function $requestFulfilled$ checks if the module that will handle the request has already fulfilled the request and is defined as:

$$requestFulfilled(pos) \equiv result_r(request(pos)) \neq \texttt{undef}$$

When the answer for the request from the module is ready, the location $result_r$ for the request will contain the computed values. The number of locations in the rightmost portion of the **request** rule is not constrained by the number of expected output parameter of the correspondent entry. This permit the specification of fewer parameter if the module client is not interested in all the results produced by the module service. The selection of which parameters to take is bound to the positional occurrence in the entry definition[2]. If the requester specify a number of location parameter greater than the entry definition output parameters number, the exceeding location will be set to `undef`.

Asynchronous Service Request

The default behaviour of the request mechanism keeps the the evaluation of the **request** rule at the same position of the syntax tree until the request is not fulfilled. In this sense the evaluation of a request is stuck until the module performs and completes it. In this section we address the scenario in which a non blocking version of **request** is desirable. To achieve the asynchronous

[2]Another way to select parameter would be by assigning a selector like $l \leftarrow x$ where l is the location in which the requester wants the parameter named x in the entry definition.

behaviour for service requests there are different possible approaches. One possibility is to spawn a new agent that performs the request. This way the agent program will contain the request and the spawner machine can continue its computation. Some synchronisation will be necessary on the request completion. Another way to asynchronously request a service is to provide a new transition rule that passes also the rule to be called on completion into the request in a sort of continuation passing style. This approach would make specification more intricate and less readable. A third approach requires the introduction of a new asynchronous transition rule for asynch request that produces an empty update set and the newly created request as value. A set of predicates in the form of expressions on request elements will allow the requester to assert on the request features, in particular the completion state. The following rules patterns will show the semantic of asynchronous requests along with an expression that asserts the completion of the request.

$$\frac{[\![t]\!]^S_\zeta \in \textsc{Module} \quad e \in entries_e([\![t]\!]^S_\zeta) \quad name_e(e) = x \quad r \in requests_r([\![t]\!]^S_\zeta)\ entry_r(r) = e\ params_r(r) = \langle [\![t_1]\!]^S_\zeta, \ldots, [\![t_n]\!]^S_\zeta \rangle}{[\![\textbf{asynch request } t \textbf{ to } x(x_1, \ldots, x_n) \textbf{ in } x']\!]^S_\zeta = \{(x', r)\}}$$

The related semantic for the CoreASM interpreter is the following:

Asynchronous service request

```
(| asynch request ^α□ to ^β x(^λ1□, ..., ^λn□) |) →
        pos := α

(| asynch request ^α v to ^β x(^λ1□, ..., ^λn□) |) →
        if v ∉ Module then
          Error("Requesting to non module")
        else
          choose e in entries_m(^α v) with name_e(e) = ^β x do
            choose λ ∈ {λ1, ..., λn} with ¬evaluated(λ) do
              pos := λ
            ifnone
                let r = new(Request) in
                  entry_r(r) := e
                  params_r(r) := ⟨λ1, ..., λn⟩
                  request(pos) := r
                  add r to requests_m(^α v)
                  [[pos]] := (undef, {|} , r)
```

The evaluation of an asynch request produces an empty update set and the newly created request element as value. To manage the just created request element we introduce the **completed** expression that checks if the request has been completed and evaluates to *true* if it is applied to a Request element that has its $result_r$ function defined.

$$[\![\textbf{completed } t]\!]^S_\zeta = \begin{cases} true & \text{if } [\![t]\!]^S_\zeta \in \textsc{Requests} \wedge result_r([\![t]\!]^S_\zeta) \neq \texttt{undef} \\ false & \text{otherwise} \end{cases}$$

The related semantic for the CoreASM interpreter is the following.

Completed expression

$(\!|\ \textbf{completed}\ {}^{\alpha}\boxed{\text{e}}\ |\!) \rightarrow \quad pos := \alpha$

$(\!|\ \textbf{completed}\ {}^{\alpha}\boxed{\text{e}}\ |\!) \rightarrow$ **if** ${}^{\alpha}v \notin$ REQUEST **then**
Error("Trying to assert completion of non-request element")
else
$[\![pos]\!] := (\texttt{undef}, \texttt{undef}, result_r({}^{\alpha}v) \neq \texttt{undef})$

Combining **asynch request** and **completed** expression the service request will not block the requester. When the request has been answered, the answer must be accessed to retrieve its data. In order to get the data the **answer** rule must be called. Its semantic is defined as:

$$\frac{[\![t]\!]^S_\zeta \in requests_r([\![module_m(self)]\!]^S_\zeta) \quad result_r([\![t]\!]^S_\zeta) \neq \texttt{undef} \quad \langle v_1, \ldots, v_n\rangle = result_r([\![t]\!]^S_\zeta)}{[\![\textbf{answer}\ t\ \textbf{into}\ x_1, \ldots, x_n]\!]^S_\zeta = \{(x_1, v_1), \ldots, (x_n, v_n)\}} \quad (1)$$

$$\frac{[\![t]\!]^S_\zeta \in requests_r([\![module_m(self)]\!]^S_\zeta) \quad result_r([\![t]\!]^S_\zeta) = \texttt{undef}}{[\![\textbf{answer}\ t\ \textbf{into}\ x_1, \ldots, x_n]\!]^S_\zeta = \emptyset} \quad (2)$$

The related semantic for the CoreASM interpreter is the following:

answer rule

$(\!|\ \textbf{answer}\ {}^{\alpha}\boxed{\text{e}}\ \textbf{into}\ {}^{\lambda_1}\boxed{\text{e}}, \ldots, {}^{\lambda_n}\boxed{\text{e}}\ |\!) \rightarrow$
$pos := \alpha$

$(\!|\ \textbf{answer}\ {}^{\alpha}v\ \textbf{into}\ {}^{\lambda_1}\boxed{\text{e}}, \ldots, {}^{\lambda_n}\boxed{\text{e}}\ |\!) \rightarrow$
if ${}^{\alpha}v \notin$ REQUEST **then**
Error("Trying to assert completion of non-request element")
else
choose $\lambda \in \lambda_1, \ldots, \lambda_n$ **with** $\neg evaluated(\lambda)$ **do**
$pos := \lambda$
ifnone
if $result_r(request(pos)) = \texttt{undef}$ **then**
$[\![pos]\!] := (\texttt{undef}, \{|\langle {}^{\lambda_1}l, \texttt{undef}\rangle, \ldots, \langle {}^{\lambda_m}l, \texttt{undef}\rangle|\}, \texttt{undef})$
else
let $\langle v'_1, \ldots, v'_{m'}\rangle = result_r(request(pos))$ **in**
let $k = min(m', m)$ **in**
$[\![pos]\!] := (\texttt{undef}, \{|\langle {}^{\lambda_1}l, v'_1\rangle, \ldots, \langle {}^{\lambda_k}l, v'_k\rangle, \langle {}^{\lambda_{k+1}}l, \texttt{undef}\rangle, \ldots, \langle {}^{\lambda_m}l, \texttt{undef}\rangle|\}, \texttt{undef})$

This rule assigns the values produced by the entry to the locations given as argument. It first evaluates the request expression. If it is an element in REQUEST the evaluation continues, otherwise an error is raised. The next step of the evaluation is to generate the updates for the locations that will contain the data produced by the service. The process is equal to the final section of the synchronous request rule definition. Combining **answer** and **completed** it is possible to manage and control the asynchronous requests to module services.

The expression **wait** transforms an asynchronous request into a synchronous one. Its semantic is defined as follow:

$$\frac{[\![t]\!]^S_\zeta \in requests_r([\![module(self)]\!]^S_\zeta) \quad result_r([\![t]\!]^S_\zeta) \neq \texttt{undef}}{[\![\textbf{wait}\ t]\!]^S_\zeta = \emptyset}$$

Wait expression

$(\!|\,\textbf{wait}\ {}^{\alpha}\square\,|\!) \rightarrow \quad pos := \alpha$

$(\!|\,\textbf{wait}\ {}^{\alpha}v\,|\!) \rightarrow$ **if** ${}^{\alpha}v \notin$ REQUEST **then**
Error("Trying to wait on non-request element")
else
if $result_r({}^{\alpha}v) \neq$ `undef` **then**
$[\![pos]\!] := (\texttt{undef}, \{|\,|\}, \texttt{undef})$

An example of usage of the asynchronous request is the following in which a module is requested to perform the *doComplexOperation* service asynchronously. The request is managed by an if statement that checks if there are no pending requests. If *noPendingRequests* is *true* a request is sent to the module m. Each step of the machine now checks if the request has been performed and when it is completed the answer is stored in the x location.

if *noPendingRequests* **then**
$r :=$ **asynch request** m **to** *doComplexOperation*
else
if completed r **then**
answer r **into** x

To improve compactness of asynchronous requests we provide also a version of it that calls a rule when the request has been completed: **asynch request when done**. Such rule request asynchronously a service to a module and, on completion, calls a given rule. A possible syntax would be

To improve compactness of asynchronous request is possible to provide a continuation passing style inspired version of it. For example writing something like

$$\textbf{asynch request}\ m\ \textbf{to}\ {}^{\beta}e(x_1, \ldots, x_n)\ \textbf{into}\ y_1, \ldots, y_m\ \textbf{when done}\ \ r$$

the request to m for an entry e would still be asynchronous but, when the request has been performed, the evaluation of the rule r is triggered. We do not give the formal semantic of this rule but it can be defined as a composition of the asynchronous requests management rules we have provided.

Accepting requests

In order to perform a service, a module must manage its requests. We introduce the new transition rule **accept** that tries to pick a service request in the pool of pending requests of a module and performs the associated service. The semantic of the **accept** rule is defined as:

$$\frac{[\![\varphi]\!]^S_\zeta = false}{[\![\textbf{accept}\ x(x_1, \ldots, x_n)\ \textbf{with}\ \varphi\ \textbf{do}\ P]\!]^S_\zeta = \emptyset} \quad (1)$$

$$\frac{[\![\varphi]\!]^S_\zeta = true \quad [\![module(self)]\!]^S_\zeta = m \quad m \in \text{MODULE} \quad \forall r \in requests_r(m). \exists e \in entries_m(m) : name_e(e) = x \wedge entry_r(r) = e}{[\![\textbf{accept}\ x(x_1, \ldots, x_n)\ \textbf{with}\ \varphi\ \textbf{do}\ P]\!]^S_\zeta = \emptyset} \quad (2)$$

$$\frac{[\![\varphi]\!]^S_\zeta = true \quad r \in requests_m(m) \quad [\![module(self)]\!]^S_\zeta = m \quad name_e(e) = x \; module_e(e) = m \; m \in \textsc{Module} \quad e \in \textsc{Entry} \; [\![P \frac{v_1}{x_1} \ldots \frac{v_n}{x_n}]\!]^S_\zeta = U}{[\![\textbf{accept } x(x_1, \ldots, x_n) \textbf{ with } \varphi \textbf{ do } P]\!]^S_\zeta = U} \quad (3)$$

The related semantic for the CoreASM interpreter is the following:

Accept rule semantic

```
(| accept ^α x(^λ1 x, ..., ^λn x) with ^β □ do ^γ □ |) →  pos := β

(| accept ^α x(^λ1 x, ..., ^λn x) with ^β v do ^γ □ |) →
    if ^β v = true then
      if workCopy(γ) = undef then
        let m = module(executingAgent) in
          choose e in entries_m(m) with name_e(e) = ^α x do
            choose r ∈ requests_m(m) with entry_r(r) = e do
              remove req from requests_m(m)
              let c = CopyTreeSub(γ, ⟨^γ1 x, ..., ^γn x⟩, params(r)) in
                pos := c
                workCopy(γ) := c
                parent(c) := pos
                request(pos) := req
            ifnone
              [[pos]] := (undef, {|}, undef)
          ifnone
            Error("Entry not available in module")
      else
        [[pos]] := (undef, updates(workCopy(γ), value(workCopy(γ))
        workCopy(γ)
        result_r(request(pos)) := [v1, ..., vn | ∀i ∈ {1, ..., n}.⟨^λi x, vi⟩ ∈ ^workCopy(γ) u ∨ vi = undef]
    else
      [[pos]] := (undef, {|}, undef)
```

If multiple **accept** for the same entry are executed in parallel, one can control their execution by their guards. The execution of more than one accept statement for the same entry and the same request is allowed when their execution produces consistent update sets. The **with** condition guards the execution of an **accept** rule. As syntactic sugar the guard can be omitted. An **accept** rule without the guard is considered having the guard always satisfiable. Another assumption for the **accept** semantic is that the entry names are unique for a module and that there are no overloaded definition for an entry.

We have decided to separate the **accept** behaviour definition from the entry in order to permit different behaviours for the same entry. Service management becomes in this way context dependent. For example it is possible to define a different behaviour for the same entry based on the module state.

Notice that a guarded **accept** rule is different from an **accept** with an if statement inside its body. For example

```
accept x with A do R1
accept x with B do R2
```

is different from

```
accept x do
  if A then R1
  if B then R2
```

because in the first case no request is consumed if A or B are not satisfied while, in the second case, a request to the service x is selected and consumed.

5.4 Examples

In this section we give some example of modules and service requests. We start with the definition of a simple echo server, then we propose the specification of a data structure where the logic of its behaviour is described by the module behaviour. Finally we give an example for a simplified Network File System.

5.4.1 Echo Service

A echo service is a server that exposes a single service *echo* that takes a parameter and return it to the service requester as result. We define a module as:

Echo service module

```
module Echo with
  entry echo(x) into y
  do
    accept echo(x) do
      y := x
```

Before requesting the module to perform its service we first have to import it and then use the **request** rule. In this example we assume that an echo module has been imported into the location $echoModule$.

Using the echo module

```
request echoModule to echo("Hello!") into answer
```

5.4.2 Data structures

Another example in which the module we defined is useful is in expressing data structures and their operations. In this examples we show a module that specifies a buffer and a simple Network File System (NFS).

N-Buffer

A buffer is finite sequence of elements. We want to describe a buffer of n position as a module that exposes five services: buffer initialisation (*init*), element insertion (*insert*), element removal (*consume*), buffer *clear*ing and *reset*. We define the buffer state as a queue.

Buffer module

```
module Buffer with
  entry init(n)
  entry insert(item)
  entry consume into res
  entry clear
  entry reset
  do
    if state = NotInitialized then
      accept init(n) do
        queue = []
        maxLength := n
        state = Initialized

    if state = Initialized then
      accept insert(item) with |queue| < maxLength do
        enqueue item into queue
      accept consume with |queue| > 0 do
        dequeue res from queue
      accept clear do
        queue = []
      accept reset do
        state = NonInitialized
```

The module is composed by five **entry** elements. A buffer can be in two states (*Initialized*, *NonInitialized*). In the *NonInitialized* state only the *init* entry can be served since we are assuming that the buffer has to be initialised with a specified capacity before using it. After initialisation the module state becomes *Initialized* and all the buffer services except for init are available. This is only one of the possible definitions for buffer behaviours and the module body is meant to provide such behaviour description while the entries expose the service interface. The module body is implemented by a control state ASM where *state* is the mode[3].

5.4.3 NFS

In this example we want to provide a module that supplies Network File System (NFS) services. In our case a NFS exposes entries for operation on files, in particular *lookup* to get a file handle from its name, *open*, *read*, *write*, *close* to provide the access to files. Since we do not want to give a full definition of a NFS behaviour, for clarity we delegate the technical details of how to get a file from its name of how to read it to some ASM rules that we assume to be defined and that provide the expected behaviour. These rules are Find($name$) to look up a file by name returning a file handle, ReadFile($file, count, offset$) and WriteFile($file, data, offset$) to respectively read and write some data at a given offset, finally Close($file$) for closing a file.

[3]This module behaviour could be also specified by the **do**-block constructs we describe in Chapter 4

Managing the access to files

```
module NFS with
  entry lookup(name) into handle
  entry read(file, count, offset) into data
  entry write(file, data, offset) into res
  entry close(file) into closed
  do
    accept lookup(name) do
      handle ← Find(name)

    accept read(file, count, offset) with canRead(file) do
      data ← ReadFile(count, offset)

    accept write(file, data, offset) with canWrite(file) do
      res ← WriteFile(file, data, offset)

    accept close(file) with isOpen(file) do
      res ← Close(file)
```

The services are provided with guards. The *isOpen* function evaluates to true if the file parameter has been opened. The other two functions, *canRead* and $canWrite$, are defined as follows:

$$canRead(file) = isOpen(file) \wedge mode(file) = read$$
$$canWrite(file) = isOpen(file) \wedge mode(file) = write$$

The module behaviour is the parallel execution of the **accept** rules for its entries. Nevertheless, the accept guards, implicitly, provide a simple workflow for the services.

Chapter 6

Modularity of State

In this chapter we will to explore the management of the ASM state. At the core of the ASM language there are two postulates [44] that are related to states: sequential and abstract state. The sequential state postulate asserts that the behaviour of a sequential time algorithm is determined by the set of states, the subset of initial states, and the state transition function. The abstract state postulate asserts that states of an algorithm are structures of a fixed vocabulary. Based on these postulates, an ASM describe any computation as the evolution of states in discrete time steps. In ASM models the state is global but as stated in [12] at chapter four

> The characteristics of basic ASMs – simultaneous execution of multiple atomic actions in a global state – come at a price, namely the lack of direct support for practical composition and structuring principles.

For this reason basic ASMs have been enriched with Turbo ASMs introducing sequential composition, iteration and (recursive) parameterized submachines. Those extensions provide submachine calls mechanism for black box computations. One of the structuring principles that Turbo ASMs do not directly provide is state partitioning that in object oriented programming [78] becomes one of the characterising features: encapsulation.

With regard to encapsulation, the ASM offer a basic construct to isolate a portion of the state with *local* declaration of some dynamic function (see chapter 2). Local functions are part of the TurboASM extension and allows functions to be declared as local to a rule, so the scope of local functions is the rule in which they are defined. Each local function f_i may be initialised by an $Init_i$ rule that is executed before the *body*.

Local definition

$name(x_1, \ldots, x_n) =$
  local $f_1[Init_1] \ldots f_k[Init_k]$
  body

Figure 6.1: Local function definition

Usually this scoping is not enough for describing a subset of the state that shares some concept meaningful for the problem domain of a specification. Some function should be shared among different rules but not globally.

Location parameterization is the most common approach that years of modelling with ASM produced to overcome the state partition problem. A concrete example could be the description of some business process that involves different organisations; one of the requirements may need the separation of information between such organisation. Being able to isolate the state by organisation is one way to model this requirement. So in the business process example each function that belongs

only to one organisation will have the organisation as first parameter. In the following example a function models the set of *employees* for each organisation.

Employee example

FinancialDepartment$(invoice) \equiv$
  choose $e \in employees$ **do**
    $ProcessInvoice(e, invoice)$

Depot$(order) \equiv$
  choose $e \in employees$ **do**
    $CheckOrder(e, order)$

In this example we assume to have two organisations (the *Depot* and the *FinacialDepartment*) that belong to the domain we are modelling. Each of the organisations needs to manage its employees but for different purposes. Here we simplify showing just one process for each organisation. The financial department needs his employees to process invoices, while the depot requires that its employees to check the orders they receive. The location *employees* intuitively contains a set of employees.

Since the two organisation are different but both of them share the concept of *employee*, in a global state view we cannot assume that the same location holds the information about two different sets of employees. There are multiple possibilities to discriminate these sets. The simplest one is to assume to have a static function $worksAt : \textsc{Organisation} \times \textsc{Employee} \rightarrow \textsc{Boolean}$ that given an organisation and an employee returns true if the employee works at the organisation. The above employee selections will become:

$$\textbf{choose } e \in employees \textbf{ with } worksAt(o, e) = true \textbf{ do } R$$

Where $o \in \textsc{Organisation}$ is an element of the Organisation domain holding the currently needed organisation (depot or financial department).

One problem of this approach relates to the static function *worksAt* that describes a static set of employees and so does not allow modifications of the employee set. This could be a problem if the set of employees may change to reflect, for example, hiring processes.

The common way to handle this situation is to rely on function parameterization. For each function that should belong to a certain scope, it is parameterized adding as first parameter such scope for each occurrence of it. The typical example of this is the parameterization with the agent running a rule that contain such functions with the *self* parameter. With regard to the business process example we need to add the organisation parameter to the *employees* location. The example would become:

Employee example

FinancialDepartment$(invoice) \equiv$
  choose $e \in employees(\text{“fd”})$ **do**
    $ProcessInvoice(e, invoice)$

Depot$(order) \equiv$
  choose $e \in employees(\text{“depot”})$ **do**
    $CheckOrder(e, order)$

In this case, the function *employees* is defined for the parameters *fd* and *depot*. This has the same effect of having two separate functions named, for example, $fdEmployees$ and $depotEmployees$, but this approach keeps the concept of “set of employees” in the same place (the employees). The downside of parameterising the locations this way is the burden for the modeller that must keep all the occurrences of interested locations up to date every time she changes something related.

Is possible to think of other possibilities to partition the state different from location parameterization or suppose that state parameterization is not general enough to manage state partition properly.

One choice to describe state partition could be to add the semantics related to the evaluation environment, the ambient, changing the foundational semantic of ASMs, in particular the update definition. An update is a couple (l, v) where l is a location and v is the value to associate to that location in the state. A location l is as well a couple $\langle f, \bar{p} \rangle$ defining for which function name f and at which point $\bar{p}$ the update is applied. Adding the ambient information, to describe the state partition, would result into a triple in the form $\langle f, e, \bar{p} \rangle$ where e represents the ambient in which the location is evaluated and updated. However this approach is not more general than location parameterization. In fact it is possible to find a bijective function that transforms the update from the changed semantic version to the parameterized one and vice versa. An example of such function is the following

$$\langle f, e, \bar{p} \rangle \leftrightarrow \langle f, e \cdot \bar{p} \rangle$$

where the environment parameter is concatenated to the other function parameters. This shows that the parameterized version, without changing the semantic, encompasses the modified semantic version of ambient support for ASMs.

Another approach would be to change the function part of the $\langle f, \bar{p} \rangle$ couple. For example having as first argument a function value instead of a function name. The function value evaluation would depend on the desired ambient. Also in this case it suffices to define a function that given a function name returns the correspondent function value to go back to the classic update definition.

Thus location parameterization is a general enough mechanism to handle state partition. In fact, recently the ambient ASMs have been proposed [17] initially to provide support for programming practices (such as object-oriented design patterns, and static and dynamic disciplines for state isolation). The ambient ASMs can be instantiated to any environment paradigm. A set of examples mostly taken from design patterns [41] and notably the ambient logic by Cardelli et al.[26]. In term of ambient ASMs the previous *Employee* example would be written:

Employee example with ambient ASMs

```
FinancialDepartment(invoice) ≡
  amb "fd" in
    choose e ∈ employees do
      ProcessInvoice(e, invoice)

Depot(order) ≡
  amb "depot" in
    choose e ∈ employees do
      CheckOrder(e, order)
```

By the definition of the *term* and *rule translation* of ambient ASM definition, this example is translated to:

Employee example equivalent to ambient ASMs

```
FinancialDepartment(invoice) ≡
  let curamb = "fd" in
    choose e(curamb) ∈ employees(curamb) do
      ProcessInvoice(e(curamb), invoice)

Depot(order) ≡
  let curamb = "depot" in
    choose e(curamb) ∈ employees(curamb) do
      CheckOrder(e(curamb), order)
```

Basically each **amb** rule occurrence is translated to a **let** rule assigned to the correct ambient value and each subterm is changed following some translation rules.

Although the ambient ASMs are general enough for describing each possibly needed environment paradigm (example of lexical and dynamic scoping are given throughout the article), **amb**

results impractical for modelling purposes. To understand why we think **amb** is not the most comfortable method to model environment aware ASMs we provide a motivating example.

Let us consider nested ambients. The default behaviour of **amb** is to consider as ambient the last computed *curamb*.

nested ambients

amb e_1 **in**
  amb e_2 **in**
    R

So the rule R will be evaluated in an environment with ambient equals to e_2. Usually the evaluation of nested ambients results in the last ambient expression as current ambient. This behaviour is not what is always desirable. Sometime having a structure on the evaluated ambient may help express clearly some state isolation. Sticking with the organisation example we now want to describe the employees of the depot but separating the clerks from the engineers. A clear way to express this is hierarchical structure of the nested ambients.

Depot employees separation

amb *depot* **in**
  amb *clerks* **in**
    $employees := \{c_1, \ldots, c_h\}$
  amb *engineers* **in**
    $employees := \{e_1, \ldots, e_k\}$

The intended meaning of such rule is to set the location *employees* keeping the worker roles distinct. With the ambient ASMs will be translated to the following equivalent version.

Depot employees separation translation

let $curamb = depot$ **in**
  let $curamb = clerks$ **in**
    $employees(curamb) := \{c_1, \ldots, c_h\}$
  let $curamb = engineers$ **in**
    $employees(curamb) := \{e_1, \ldots, e_k\}$

The evaluation of such rule will produce the same update set of

$employees(clerks) := \{c_1, \ldots, c_h\}$
$employees(engineers) := \{e_1, \ldots, e_k\}$

While the intended meaning was closer to

$employees(depot, clerks) := \{c_1, \ldots, c_h\}$
$employees(depot, engineers) := \{e_1, \ldots, e_k\}$

The difference becomes meaningful if we want *employees* to contain the information about the employees other organisations (e.g. financial department $employees(fd, clerks) := v$).

Another issue in using the ambient ASMs is related to identifiers evaluation. We recall here one of the examples in [17].

Evaluate identifier with ambient

Example$_2 \equiv$
  amb a_1 **in**
    $x := 3$
    amb a_2 **in**
      $y := parent(a2).x$

This example is meant to show the need of expressing explicitly an ambient where to evaluate an expression (x). The function *parent* should reflect the nesting of occurrences for **amb** rules; in this example $parent(a_2) = a_1$. The first issue with this solution is syntactical: accessing ambients from a deeply nested position quickly makes the evaluation unreadable.

evaluating in deeply nested ambients

amb a_1 **in**
  $x := 3$
  amb a_2 **in**
    ...
      amb a_k **in**
        $y := parent(parent(parent(\ldots parent(a_2))).x$

The second issue is related to the capability of accessing to the value of an identifier evaluated in an arbitrary ambient. The *parent* functions only allows to climb back the nested structure of ambients. So we consider impractical using ambient ASMs extensively to replace common patterns of ASM modelling.

In this chapter we introduce an extension to basic ASMs that allows the management of environments to partition the state and ease the modelling process. In particular we propose three kind of state partition management that are covered by three transition rules and one evaluation expression for ambients. We explore the three possibilities and propose one unified version that combines the three versions.

6.1 State partition management

We identify three interesting cases of environment management for state locations of an ASMs: plain, hierarchical and set ambients. Plain ambient allows to assign an environment to a rule block. The behaviour is equivalent to ambient ASMs, an expression is used as environment selector and the location inside the rule block are evaluated is such environment. With hierarchical ambient, environments are pushed following the lexical arrangement in the specification. Finally the set ambient manages environments as sets inserting and removing environment expressions based on lexical occurrence of ambient blocks but composing them without order dependence. All the ambient rules we introduce share the syntax shown in Figure 6.2.

ambient syntax

Ambient := **ambient** [*AmbientKind*][**for** *Ids*] **in** *Rule*
AmbientKind := **set** | **push** | **insert**
Ids := x | x, *Ids*

Figure 6.2: Ambient rules BNF

The *AmbientKind* **set**, **push** and **insert** correspond respectively to plain, hierarchical and set state partitioning. The intuitive meaning of a **ambient** block is to partition the state accordingly to its *AmbientKind* for each identifier inside the rule block or, if the **for** clause is specified, only for the specified identifiers leaving the others with the last ambient available. An **ambient** rule without a specified ambient kind is equivalent to a **ambient set** by default. Providing state partitioning management functionalities for the ASM framework requires to define a rule for the evaluation of identifiers that is aware of environment definition, and to allow their manipulation. For the first requirement we provide an evaluation rule definition that is a conservative extension of the one in [37]. We choose this approach because we recall that we want to provide construct that are easily implementable and nimbly integrated with the existing tool support for ASMs, especially *CoreASM*. For the requirement on environment manipulation we have chosen to annotate the abstract syntax tree (AST) of the evaluating **ambient** rule with information about ambients,

and to use working copies of rules inside the ambient block. Choosing to use working copies of AST trees enables our rule definition to handle uniformly location parameterization in case of rule calls inside the ambient rule block. An example is given by the parallel execution of the following **ambient** rules:

rule calls inside ambient rules

```
R1 ≡ ambient 1 in
       seq
         x := new(K)
       next
         M

R2 ≡ ambient 2 in
       seq
         x := new(K)
       next
         M

M ≡ ambient x in R3
```

The evaluation of M must be aware of the contextual scope for x accordingly to where M is called (1 for R_1, 2 for R_2). Using the work copy is also a consequence of abiding to the CoreASM interpreter semantic.

In the following sections we give the semantic for each of the ambient rule kind we propose and some examples.

6.1.1 Assigning ambients

The first kind of state partition we address is the *ambient assignment*. This ambient version is expressed by the **set** option for *AmbientKind* production in fig 6.2. The intuitive meaning is to set the evaluation environment for each identifier in the ambient rule part to the ambient expression (e) value. If an environment was already defined for such identifier, it is replaced by the new one. The formal semantic for **ambient set** is the following:

Assign scope rule

$$(\!|\ \textbf{ambient set } {}^{\alpha}\boxed{\text{c}} \textbf{ in } {}^{\beta}\boxed{\text{r}}\ |\!) \rightarrow \quad pos := \alpha$$

$$(\!|\ \textbf{ambient set } {}^{\alpha}v \textbf{ in } {}^{\beta}\boxed{\text{r}}\ |\!) \rightarrow \quad \textbf{if } workCopy(\beta) = \texttt{undef} \textbf{ then}$$
$$\textbf{let } b = \mathsf{CopyTree}(\beta) \textbf{ in}$$
$$\mathsf{SetAmbient}_{set}(b, {}^{\alpha}v)$$
$$pos := b$$
$$parent(b) := pos$$
$$workCopy(\beta) := b$$
$$\textbf{else}$$
$$[\![pos]\!] := (\texttt{undef}, {}^{workCopy(\beta)}u, {}^{workCopy(\beta)}v)$$

$$(\!|\ \textbf{ambient set } {}^{\alpha}\boxed{\text{c}} \textbf{ for } {}^{\lambda_1}x, \ldots, {}^{\lambda_n}x \textbf{ in } {}^{\beta}\boxed{\text{r}}\ |\!) \rightarrow \quad pos := \alpha$$

$$(\!|\ \textbf{ambient set } {}^{\alpha}v \textbf{ for } {}^{\lambda_1}x, \ldots, {}^{\lambda_n}x \textbf{ in } {}^{\beta}\boxed{\text{r}}\ |\!) \rightarrow \quad \textbf{if } workCopy(\beta) = \texttt{undef} \textbf{ then}$$
$$\textbf{let } b = \mathsf{CopyTree}(\beta) \textbf{ in}$$
$$\mathsf{SetAmbientId}_{set}(b, {}^{\alpha}v, \langle {}^{\lambda_1}x, \ldots, {}^{\lambda_n}x\rangle)$$
$$pos := b$$
$$parent(b) := pos$$
$$workCopy(\beta) := b$$
$$\textbf{else}$$
$$[\![pos]\!] := (\texttt{undef}, {}^{workCopy(\beta)}u, {}^{workCopy(\beta)}v)$$

The first step of the rule semantic is to evaluate the environment expression in position α of the abstract syntax tree. Once the environment expression value ${}^{\alpha}v$ is available, the rule tree is copied and set as the next position to be evaluated. The $\mathsf{SetAmbient}_{set}$ rule is in charge of setting the evaluation ambient for each identifier in the evaluating rule tree. The result of the evaluation is the update set and the value computed by the work copy rule. The other two pattern definition of the semantic behave similarly to the **ambient** rule definition but take into account also a set of identifiers. The list of identifier ${}^{\lambda_1}x, \ldots, {}^{\lambda_n}x$ permits to narrow the set of identifiers on which setting the ambient is desirable: only the identifiers in the list will be considered for environment definition.

SetAmbient rule definition

$\mathsf{SetAmbient}_{set}(\alpha, env) \equiv$
  if $class(\alpha) = \mathsf{id} \wedge isLocationName(token(\alpha))$ **then**
    $locEnv(\alpha) := env$
  else
    if $isRulename(token(\alpha))$ **then**
      let $r = ruleValue(token(\alpha))$ **in**
      let $c = \mathsf{CopyTree}(body(r))$ **in**
        $\mathsf{TreeSub}(\alpha, c)$
        $parent(c) := parent(\alpha)$
        $\mathsf{SetAmbient}_{set}(c, env)$
    else
      $\mathsf{SetAmbient}_{set}(first(\alpha), env)$
      $\mathsf{SetAmbient}_{set}(next(\alpha), env)$

The rule $\mathsf{SetAmbient}_{set}$ manages environment changes for identifiers. The two parameters represent the tree reference (α) and the the environment value (env). The idea is to annotate each tree position that contains an identifier with the environment information. If the tree reference α is an identifier for a location the environment is assigned to $locEnv(\alpha)$. Otherwise if it is a rule name a copy of the rule body is created and substitutes to α; the $\mathsf{SetAmbient}_{set}$ rule is applied to it. If the current tree reference is neither an identifier nor a rule name, the $\mathsf{SetAmbient}_{set}$ rule is applied recursively to the tree children.

SetAmbientId rule definition

$\mathsf{SetAmbientId}_{set}(\alpha, env, locationNames) \equiv$
  if $class(\alpha) = \mathsf{id} \wedge isLocationName(token(\alpha)) \wedge token(\alpha) \in locationNames$ **then**
    $locEnv(\alpha) := env$
  else
    if $isRulename(token(\alpha))$ **then**
      let $r = ruleValue(token(\alpha))$ **in**
      let $c = \mathsf{CopyTree}(body(r))$ **in**
        $\mathsf{TreeSub}(\alpha, c)$
        $parent(c) := parent(\alpha)$
        $\mathsf{SetAmbientId}_{set}(c, env, locationNames)$
    else
      $\mathsf{SetAmbientId}_{set}(first(\alpha), env, locationNames)$
      $\mathsf{SetAmbientId}_{set}(next(\alpha), env, locationNames)$

The $\mathsf{SetAmbientId}_{set}$ rule behaviour is similar to $\mathsf{SetAmbient}_{set}$ but the only identifiers affected by environment replacement are the set denoted by the third parameter ($locationNames$).

We now show some properties of this kind of state partitioning with **ambient set** .

Property 6.1.1. (Nested **ambient set**) Two or more nested occurrence of the **ambient set** transition rule applied to a given rule R with the same environment expression e are equivalent to a single **ambient set** rule. So writing:

---

ambient set e **in**
  ambient set e **in**
    R

---

is equivalent to:

---

ambient set e **in**
  R

The second property is a generalisation of the first. Both of them derives from the replacement properties of the assignment.

Property 6.1.2. (Environment replacement) Nested **ambient set** rules with $e_1, \ldots, e_n$ environment expression is equivalent to a single **ambient set** with the innermost environment expression e_n. Writing

---

ambient set e_1 **in**
  ambient set e_2 **in**
    ...
    ambient set e_n **in**
      R

---

is equivalent to

---

ambient set e_n **in**
  R

6.1.2 Hierarchical ambients

With hierarchical ambients we want to cover and ease one of the most common strategy that leverages multiple parameter to partition the state of locations. While plain ambient assignment (see Section 6.1.1) just replace the evaluation environment for identifiers, hierarchical ambients manages the current environment keeping track of the previously occurred environment expressions. By simplifying location parameterization the readability of specifications improves. Intuitively nesting environment expression corresponds to adding one parameter for each environment value to each function. So for example:

---

ambient push e_1 **in**
  ambient push e_2 **in**
    $x := 3$

---

should produce the location update $(x, \langle e_1, e_2 \rangle, 3)$ that is equivalent to the location assignment

---

$x(e_1, e_2) := 3$

The advantage of using **ambient** instead of simple assignments becomes more interesting when the number of parameters and locations with multiple parameters increases. The semantic for **ambient push** is defined as:

PushAmbient

$(\!|\,\textbf{ambient push } {}^{\alpha}\boxed{\text{e}} \textbf{ in } {}^{\beta}\boxed{\text{r}}\,|\!) \rightarrow \quad pos := \alpha$

$(\!|\,\textbf{ambient push } {}^{\alpha}v \textbf{ in } {}^{\beta}\boxed{\text{r}}\,|\!) \rightarrow$ **if** $workCopy(\beta) = \texttt{undef}$ **then**
  let $b = \mathsf{CopyTree}(\beta)$ **in**
    $\mathsf{SetAmbient}_{push}(b, {}^{\alpha}v)$
    $pos := b$
    $parent(b) := pos$
    $workCopy(\beta) := b$
else
  $[\![pos]\!] := (\texttt{undef}, {}^{workCopy(\beta)}u, {}^{workCopy(\beta)}v)$

$(\!|\,\textbf{ambient push } {}^{\alpha}\boxed{\text{e}} \textbf{ for } {}^{\lambda_1}x, \ldots, {}^{\lambda_n}x \textbf{ in } {}^{\beta}\boxed{\text{r}}\,|\!) \rightarrow \quad pos := \alpha$

$(\!|\,\textbf{ambient push } {}^{\alpha}v \textbf{ for } {}^{\lambda_1}x, \ldots, {}^{\lambda_n}x \textbf{ in } {}^{\beta}\boxed{\text{r}}\,|\!) \rightarrow$ **if** $workCopy(\beta) = \texttt{undef}$ **then**
  let $b = \mathsf{CopyTree}(\beta)$ **in**
    $\mathsf{SetAmbientId}_{push}(b, {}^{\alpha}v, \langle {}^{\lambda_1}x, \ldots, {}^{\lambda_n}x \rangle)$
    $pos := b$
    $parent(b) := pos$
    $workCopy(\beta) := b$
else
  $[\![pos]\!] := (\texttt{undef}, {}^{workCopy(\beta)}u, {}^{workCopy(\beta)}v)$

The environment expression α is evaluated and the rule body β is copied creating the *workCopy* that will produce the updates of the partitioned location inside the rule. The *workingCopy* is annotated by the $\mathsf{SetAmbient}_{push}$ rule and the evaluation position is set to the working copy. The result of the evaluation is the update set produced by the value of the interpretation of $workingCopy(\beta)$. The second form of **ambient push** annotates only the identifiers specified by ${}^{\lambda_1}x, \ldots, {}^{\lambda_n}x$ calling the $\mathsf{SetAmbientId}_{push}$ rule.

SetAmbientId_push rule definition

$\mathsf{SetAmbientId}_{push}(\alpha, env) \equiv$
  if $class(\alpha) = \mathsf{id} \wedge isLocationName(token(\alpha))$ **then**
    if $locEnv(\alpha) \neq \texttt{undef}$ **then**
      $locEnv(\alpha) := [env]$
    else
      $locEnv(\alpha) := locEnv(\alpha) + [env]$
  else
    if $isRulename(token(\alpha))$ **then**
      let $r = ruleValue(token(\alpha))$ **in**
      let $c = \mathsf{CopyTree}(body(r))$ **in**
        $\mathsf{TreeSub}(\alpha, c)$
        $parent(c) := parent(\alpha)$
        $\mathsf{SetAmbient}_{push}(c, env)$
    else
      $\mathsf{SetAmbient}_{push}(first(\alpha), env)$
      $\mathsf{SetAmbient}_{push}(next(\alpha), env)$

The $\mathsf{PushAmbient}_{push}$ rule firstly checks if the first parameter α (representing the tree to be annotated) is a location identifier. If so the *locEnv* location is initialised with the *env* value passed as parameter. We represent the nesting structure of **ambient** occurrences with a list of environment values, so the initialisation creates a list with a single element containing the *env* parameter. In case an ambient has already been set, the rule updates it adding the *env* to the tail of the current *locEnv*. If the first parameter is not an identifier the rule checks it to be a rule name. If a rule name is encountered, its body is copied and substituted ($\mathsf{TreeSub}$) unfolding the macro substitution. The copied tree is annotated as well with the *env* parameter. Finally, if α is

neither a location identifier nor a rule name, its children are annotated by $\mathsf{SetAmbient}_{push}$.

PushAmbientId rule definition

$\mathsf{SetAmbientId}_{push}(\alpha, env, locationNames) \equiv$
  if $class(\alpha) = \mathsf{id} \wedge isLocationName(token(\alpha)) \wedge token(\alpha) \in locationNames$ **then**
    if $locEnv(\alpha) \neq$ `undef` **then**
      $locEnv(\alpha) := [env]$
    else
      $locEnv(\alpha) := locEnv(\alpha) + [env]$
  else
    if $isRulename(token(\alpha))$ **then**
      let $r = ruleValue(token(\alpha))$ **in**
      let $c = \mathsf{CopyTree}(body(r))$ **in**
        $\mathsf{TreeSub}(\alpha, c)$
        $parent(c) := parent(\alpha)$
        $\mathsf{SetAmbientId}_{push}(c, env, locationNames)$
    else
      $\mathsf{SetAmbientId}_{push}(first(\alpha), env, locationNames)$
      $\mathsf{SetAmbientId}_{push}(next(\alpha), env, locationNames)$

For the **ambient push** rule variant that considers a subset of identifiers, $\mathsf{SetAmbientId}_{push}$ has the same role of $\mathsf{SetAmbient}_{push}$ rule but it also checks that the location identifiers belongs to the set of interest passed as third parameter[1].

The ambient push rules hold the following property:

Property 6.1.3. (Parallel Push Environment) Parallel occurrences of **ambient push** with the same environment expression are equivalent to a single **ambient push** for such expression and as body rule the parallel block of the body rules in the original **ambient push** rules. Writing

ambient push e **in**
  R_1
ambient push e **in**
  R_2
...
ambient push e **in**
  R_n

is equivalent to

ambient push e **in**
  R_1
  R_2
  ...
  R_n

6.1.3 Set ambients

The third kind of state partition management we show manages environment as sets. With this version we address the cases in which nesting ambient rules should produces environments that are independent on their lexical order. We introduce the transition rule **ambient insert** . As an example we can consider a note sharing system in which different users can share information. We want to describe the set of shared notes among users with the location *notes*.

[1]Here we are trying to give a uniform representation of the three ambient definition to make their combination comfortable as we will describe later

```
R(user, shareWith) ≡
  ambient insert user in
    let n = new(Note) in
      FillNote(n)
      ambient shareWith in
        add n to notes

P ≡
  R(A, B)
  R(B, A)
```

In this example we are assuming to have the information about the other user with whom to share the note n in *shareWith*. Having two users A and B executing P will result in filling the *notes* location with two new notes that are shared between A and B. To access the set of shared notes it is possible to write something like:

```
ambient set A in
  ambient set B in
    forall n ∈ notes do
      ShowNote(n)
```

Where ShowNote is a rule that displays appropriately a note.

Similarly to **ambient set** and **ambient push** , evaluating the **ambient insert** rule requires first to evaluate the environment expression. After that its value is passed to $\mathsf{SetAmbient}_{insert}$ rule that annotates a work copy of the **ambient insert** body and the work copy is evaluated. The result of the evaluation is the update set and value of *workCopy*. The semantic for **ambient insert** is the following:

Set Ambient

$$(\!|\textbf{ambient insert } {}^{\alpha}\boxed{\text{e}} \textbf{ in } {}^{\beta}\boxed{\text{r}}|\!) \rightarrow \quad pos := \alpha$$

$$(\!|\textbf{ambient insert } {}^{\alpha}v \textbf{ in } {}^{\beta}\boxed{\text{r}}|\!) \rightarrow \quad \textbf{if } workCopy(\beta) = \texttt{undef} \textbf{ then}$$
$$\textbf{let } b = \mathsf{CopyTree}(\beta) \textbf{ in}$$
$$\mathsf{SetAmbient}_{insert}(b, {}^{\alpha}v)$$
$$pos := b$$
$$parent(b) := pos$$
$$workCopy(\beta) := b$$
$$\textbf{else}$$
$$[\![pos]\!] := (\texttt{undef}, {}^{workCopy(\beta)}u, {}^{workCopy(\beta)}v)$$

$$(\!|\textbf{ambient insert } {}^{\alpha}\boxed{\text{e}} \textbf{ for } {}^{\lambda_1}x, \ldots, {}^{\lambda_n}x \textbf{ in } {}^{\beta}\boxed{\text{r}}|\!) \rightarrow \quad pos := \alpha$$

$$(\!|\textbf{ambient insert } {}^{\alpha}v \textbf{ for } {}^{\lambda_1}x, \ldots, {}^{\lambda_n}x \textbf{ in } {}^{\beta}\boxed{\text{r}}|\!) \rightarrow \quad \textbf{if } workCopy(\beta) = \texttt{undef} \textbf{ then}$$
$$\textbf{let } b = \mathsf{CopyTree}(\beta) \textbf{ in}$$
$$\mathsf{SetAmbientId}_{insert}(b, {}^{\alpha}v, \langle {}^{\lambda_1}x, \ldots, {}^{\lambda_n}x \rangle)$$
$$pos := b$$
$$parent(b) := pos$$
$$workCopy(\beta) := b$$
$$\textbf{else}$$
$$[\![pos]\!] := (\texttt{undef}, {}^{workCopy(\beta)}u, {}^{workCopy(\beta)}v)$$

A variant that restricts the ambient annotation for a set of identifier is provided. Similarly to this variant that does not change the ambient of identifiers not in the list ${}^{\lambda_1}x, \ldots, {}^{\lambda_n}x$ others are possible. For example a rule that changes the ambient of all the identifiers *but* the members of the provided list. Another variant could take into account some property of the identifiers and use it

to select the identifiers affected by the ambient rule. Regular expression on the identifiers token could select all the tokens that match a given regex. We do not show these other variants but if someone is interested in having them she could follow the three semantics we have given here and provide a rule for environment annotation in the same way we did for $\mathsf{SetAmbientId}_*$. The definitions for $\mathsf{SetAmbient}_{insert}$ and $\mathsf{SetAmbientId}_{insert}$ are the following:

SetAmbient rule definition

$\mathbf{SetAmbient}_{insert}(\alpha, env, locationNames) \equiv$
  if $class(\alpha) = \mathsf{id} \wedge isLocationName(token(\alpha)) \wedge token(\alpha) \in locationNames$ **then**
    if $locEnv(\alpha) \neq \texttt{undef}$ **then**
      $locEnv(\alpha) := \{env\}$
    else
      $locEnv(\alpha) := locEnv(\alpha) + [env]$
  else
    if $isRulename(token(\alpha))$ **then**
      let $r = ruleValue(token(\alpha))$ **in**
      let $c = \mathsf{CopyTree}(body(r))$ **in**
        $\mathsf{TreeSub}(\alpha, c)$
        $parent(c) := parent(\alpha)$
        $\mathsf{SetAmbientId}_{insert}(c, env, locationNames)$
    else
      $\mathsf{SetAmbientId}_{insert}(first(\alpha), env, locationNames)$
      $\mathsf{SetAmbientId}_{insert}(next(\alpha), env, locationNames)$

PushAmbientId rule definition

$\mathbf{SetAmbientId}_{insert}(\alpha, env, locationNames) \equiv$
  if $class(\alpha) = \mathsf{id} \wedge isLocationName(token(\alpha)) \wedge token(\alpha) \in locationNames$ **then**
    if $locEnv(\alpha) \neq \texttt{undef}$ **then**
      $locEnv(\alpha) := \{env\}$
    else
      $locEnv(\alpha) := locEnv(\alpha) + [env]$
  else
    if $isRulename(token(\alpha))$ **then**
      let $r = ruleValue(token(\alpha))$ **in**
      let $c = \mathsf{CopyTree}(body(r))$ **in**
        $\mathsf{TreeSub}(\alpha, c)$
        $parent(c) := parent(\alpha)$
        $\mathsf{SetAmbientId}_{insert}(c, env, locationNames)$
    else
      $\mathsf{SetAmbientId}_{insert}(first(\alpha), env, locationNames)$
      $\mathsf{SetAmbientId}_{insert}(next(\alpha), env, locationNames)$

The peculiar property of the ambient rules on sets is that the order in which environments appear is irrelevant.

Property 6.1.4. (Nesting Order Independence) Given a set of environment expressions $e_1, \ldots, e_n$, for each permutation $e_{\sigma(1)}, \ldots, e_{\sigma(n)}$ of them such that $\sigma : \{1, \ldots, n\} \to \{1, \ldots, n\}$, nesting **ambient insert** rules with $e_1, \ldots, e_n$ is equivalent to nesting $e_{\sigma(1)}, \ldots, e_{\sigma(n)i}$.

6.1.4 Combining ambient rules

The three state partition strategies we presented in the previous sections have different properties and cases of use. It would be beneficial for modelling purpose to have a single rule definition that embraces all of them. With such rule each ambient strategies could be combined with the others seamlessly.

The fundamental technical difference between **ambient set**, **push** and **insert** is the representation of the environment as, respectively, a value, a list or a set. With regard to **ambient set**

ambient push e_1 **in**	**ambient push** $e_{\sigma(1)}$ **in**
R_1	R_1
ambient push e_2 **in**	**ambient push** $e_{\sigma(2)}$ **in**
R_2	R_2
...	...
ambient push e_n **in**	**ambient push** $e_{\sigma(n)}$ **in**
R_n	R_n

environments, we can uniform their representations by using either lists or sets. In the first case the $\mathsf{SetAmbient}_{set}$ rule, instead of assigning the environment expression value to the location *locEnv*, will assign a list with such value as list with a single value. Similarly assigning the singleton for the environment value uniforms *set* and *insert* scopes. In order to uniform the *push* and *insert* ambient rules there are at least two possibilities: managing *set* environments as ordered sets or delegate the management of them to the evaluation function for language expressions.

The first approach requires a canonical order for the elements in a set. Since values inside a set could be heterogeneous, an ordering between two elements may not be easy to find. To provide a set with a canonical ordering we may define a function

$$canonicalPosition : \textsc{Element} \rightarrow \textsc{Number}$$

that associate each element to a number. The canonical order will be given by the numbers ordering after applying to each environment element such function[2].

The second approach demands to the specifiers the task to build a correct expression for denoting the appropriate environment. In this case the three **ambient** rules could be condensed to just **ambient set** with the appropriate environment expression as parameter. This possibility de facto takes the biggest advantages of ambient rules apart: easing environments management. For this reason we choose to adopt the fist approach.

To uniform the behaviour of **ambient set**, **push** and **insert** we choose that *locEnv* location contains only list values. With this choice **ambient set** evaluates the environment expression to a value v and annotate the rules parsing tree filling *locEnv* with a list whose single element is the evaluation of the environment expression ($[v]$). The **ambient push** does not requires changes. The **ambient insert** replaces its $\mathsf{ScopeSet}_{insert}$ and $\mathsf{ScopeSetId}_{insert}$ with the *push* version. We could have chosen to replace *push* rules by *insert* ones, we made this choice because we consider the *push* version usage more frequent. With this choice the three scope rule can be used together.

As syntactic sugar we rewrite **ambient set** as just **ambient** making it the default ambient management strategy; **ambient insert** and **ambient push** become equivalent, the difference between them is delegated to the evaluation of identifiers. Although annotating the AST nodes with environment values is crucial to manage environments, the second half of it is making the evaluation of identifiers aware of these annotations.

6.2 Identifiers evaluation

The three ambient rules we introduced in the previous sections set the stage for state partition management adding information about environment inside the *locEnv* location. In order to evaluate properly the identifiers in the correct ambient it is needed to make identifier evaluation rules aware of such information. This step depends on the implementation of the considered ASM framework. In *CoreASM* the evaluation of identifiers is part of the kernel interpreter, and consists of two stages. The first step is to check whether the identifier refers to a local variable ($env(x)$) or not. If the identifier is not a local variable reference, the interpreter checks if it is a function names

[2]One possible implementation of this kind of function is the *hashCode* of the element

and, in case, retrieves its value. The macro HandleUndefinedIdentifier comes into play if the identifier token is unknown. The formal semantic from [37] is the following:

Identifiers evaluation in CoreASM

$$
\begin{array}{lll}
(\!|\, {}^{\alpha}x \,|\!) & \rightarrow & \textbf{if } env(x) \neq \texttt{undef} \textbf{ then} \\
& & \quad [\![pos]\!] := (\texttt{undef}, \texttt{undef}, env(x)) \\
& & \textbf{then} \\
& & \quad \textbf{if } isFunctionName(x) \textbf{ then} \\
& & \quad\quad \textbf{let } l = (x, \langle\rangle) \textbf{ in} \\
& & \quad\quad\quad [\![pos]\!] := (\texttt{undef}, \texttt{undef}, getValue(l)) \\
& & \quad \textbf{if } undefiniedToken(x) \textbf{ then} \\
& & \quad\quad \mathsf{HandleUndefinedIdentifier}(pos, x, \langle\rangle)
\end{array}
$$

This definition covers identifiers without parameters, its version for parameterized identifiers is similar:

Parameterized identifier evaluation in coreASM

$$
\begin{array}{lll}
(\!|\, {}^{\alpha}x({}^{\lambda_1}[?], \ldots, {}^{\lambda_n}[?]) \,|\!) & \rightarrow & \textbf{if } isFunctionName(x) \textbf{ then} \\
& & \textbf{choose } i \in [1, n] \textbf{ with } \neg evaluated(\lambda_i) \textbf{ do} \\
& & \quad pos := \lambda_i \\
& & \textbf{ifnone} \\
& & \quad \textbf{let } l = (x, \langle value(\lambda_1), \ldots, value(\lambda_n)\rangle) \textbf{ in} \\
& & \quad\quad [\![pos]\!] := (\texttt{undef}, \texttt{undef}, getValue(l)) \\
& & \textbf{if } undefiniedToken(x) \textbf{ then} \\
& & \quad \mathsf{HandleUndefinedIdentifier}(pos, x, \langle\lambda_1, \ldots, \lambda_n\rangle)
\end{array}
$$

In order to make these evaluation definitions aware of environment information, we have to change them. Since we want to limit the amount of changes to *CoreASM* to not be disruptive to the framework when implementing ambient capabilities, we define the new evaluation definitions for identifiers as conservative extensions of the old ones. This way the implementation may be updated using the *decorator* pattern [41]. First we uniform the two identifier evaluation definitions providing a macro **EvaluateId** that includes the two versions.

$$
\begin{array}{l}
\textbf{EvaluateId}(x, args) \equiv \\
\quad \textbf{if } args = \langle\rangle \wedge env(x) \neq \texttt{undef} \textbf{ then} \\
\quad\quad [\![pos]\!] := (\texttt{undef}, \texttt{undef}, env(x)) \\
\quad \textbf{else} \\
\quad\quad \textbf{if } isFunctionName(x) \textbf{ then} \\
\quad\quad\quad \textbf{let } \langle\lambda_1, \ldots, \lambda_n\rangle = args \textbf{ in} \\
\quad\quad\quad \textbf{choose } i \in [1, n] \textbf{ with } \neg evaluated(\lambda_i) \textbf{ do} \\
\quad\quad\quad\quad pos := \lambda_i \\
\quad\quad\quad \textbf{ifnone} \\
\quad\quad\quad\quad \textbf{let } l = (x, \langle value(\lambda_1), \ldots, value(\lambda_n)\rangle) \textbf{ in} \\
\quad\quad\quad\quad [\![pos]\!] := (\texttt{undef}, \texttt{undef}, getValue(l)) \\
\quad\quad \textbf{if } undefiniedToken(x) \textbf{ then} \\
\quad\quad\quad \mathsf{HandleUndefinedIdentifier}(pos, x, \langle\lambda_1, \ldots, \lambda_n\rangle)
\end{array}
$$

With this macro we can rewrite the evaluation definitions above as:

Identifiers evaluation in CoreASM

$$
\begin{array}{lll}
(\!|\, {}^{\alpha}x \,|\!) & \rightarrow & \mathsf{EvaluateId}({}^{\alpha}x, \langle\rangle) \\
\\
(\!|\, {}^{\alpha}x({}^{\lambda_1}[?], \ldots, {}^{\lambda_n}[?]) \,|\!) & \rightarrow & \mathsf{EvaluateId}({}^{\alpha}x, \langle\lambda_1, \ldots, \lambda_n\rangle)
\end{array}
$$

We define the macro $\mathsf{EvaluateId}_{env}$ as a conservative extension of EvaluateId and then we will give the new definition for the evaluation of identifiers applying this macro. $\mathsf{EvaluateId}_{env}$ checks if the identifier is annotated with environment information, if it is `undef`ined it means that no information

about the environment is available and the evaluation is performed by the usual EvaluateId macro. If the identifier is a function ($isFunctionName$) the environments are used as prefix parameters of it. Otherwise the old EvaluateId semantic is executed.

EvaluateId_env

$$\begin{array}{l}
\textsf{EvaluateId}_{env}(x, args, env) \equiv \\
\quad \textbf{if } env \neq \texttt{undef} \wedge isFunctionName(x) \textbf{ then} \\
\quad\quad \textbf{let } \langle \lambda_1, \ldots, \lambda_n \rangle = args \textbf{ in} \\
\quad\quad \textbf{choose } \lambda \in args \textbf{ with } \neg evaluated(\lambda) \textbf{ do} \\
\quad\quad\quad pos := \lambda \\
\quad\quad \textbf{ifnone} \\
\quad\quad\quad \textbf{let } \langle e_1, \ldots, e_k \rangle = env \textbf{ in} \\
\quad\quad\quad \textbf{let } l = (x, \langle e_1, \ldots, e_k, {}^{\lambda_1}v, \ldots, {}^{\lambda_n}v \rangle) \textbf{ in} \\
\quad\quad\quad\quad [\![pos]\!] := (\texttt{undef}, \texttt{undef}, getValue(l)) \\
\quad \textbf{else} \\
\quad\quad \textsf{EvaluateId}(x, args)
\end{array}$$

This definition is a conservative extension since

$$\textsf{EvaluationId}_{env}(x, args, \texttt{undef}) \equiv \textsf{EvaluationId}(x, args)$$

. The final definitions for identifiers evaluation becomes:

Ambient aware identifier evaluation

$$(\!|\, {}^{\alpha}x \,|\!) \rightarrow \textsf{EvaluateId}_{env}({}^{\alpha}x, \langle\rangle, locEnv(pos))$$

$$(\!|\, {}^{\alpha}x({}^{\lambda_1}\boxed{?}, \ldots, {}^{\lambda_n}\boxed{?}) \,|\!) \rightarrow \textsf{EvaluateId}_{env}({}^{\alpha}x, \langle \lambda_1, \ldots, \lambda_n \rangle, locEnv(pos))$$

Notice that the parameterization for locations in this case depends on the order of environments in the *locEnv* function. Since we also want to cover the evaluation of **ambient insert** created environments, we need to introduce a supplementary evaluation expression for them. We could have defined the identifiers evaluation expressions for sets but the ordered version is more frequent so we use it as default evaluation behaviour.

Environment sets evaluation

$$\begin{array}{lll}
(\!|\, \textbf{set eval } {}^{\alpha}x \,|\!) \rightarrow & \textbf{if } locEnv(\alpha) \neq \texttt{undef} \textbf{ then} & \\
& \quad \textsf{EvaluateId}_{env}({}^{\alpha}x, \langle\rangle, canonicalOrder(locEnv(\alpha))) & \\
& \textbf{else} & \\
& \quad \textsf{EvaluateId}({}^{\alpha}x, \langle\rangle) &
\end{array}$$

$$\begin{array}{lll}
(\!|\, \textbf{set eval } {}^{\alpha}x({}^{\lambda_1}\square, \ldots, {}^{\lambda_n}\square) \,|\!) \rightarrow & \textbf{if } locEnv(\alpha) \neq \texttt{undef} \textbf{ then} & \\
& \quad \textsf{EvaluateId}_{env}({}^{\alpha}x, \langle \lambda_1, \ldots, \lambda_n \rangle, canonicalOrder(locEnv(\alpha))) & \\
& \textbf{else} & \\
& \quad \textsf{EvaluateId}({}^{\alpha}x, \langle \lambda_1, \ldots, \lambda_n \rangle) &
\end{array}$$

The function *canonicalOrder* takes a list of environment values and returns them reordered by some canonical ordering of the elements (for example using the *canonicalPosition* function of Section 6.1.4). This way the computed list will be the same for every permutation of the elements in the scope list. We leave abstract the definition of such ordering since it is an implementation detail and can be freely defined conforming the chosen ASM framework code-base.

6.2.1 Explicit evaluation

Sometimes the lexical structure of the ambient rules must be bypassed in order to access an identifier value in a specific environment. To permit this behaviour we introduce some expressions to denote the desired evaluation environment for one identifier. We present two forms of explicit environment evaluation with two variant each for parameterless and parameterized identifiers. The first form requires an environment expression and an identifier, while the second one ignores

the environment information evaluating the identifier in the global environment (the undefined environment).

Explicit environment evaluation

$(\!|\ \textbf{eval}\ {}^{\alpha}x\ \textbf{in}\ [{}^{\lambda_1}\boxed{\text{e}}, \ldots, {}^{\lambda_n}\boxed{\text{e}}]\ |\!) \rightarrow$
  choose $i \in [1, n]$ **with** $\neg evaluated(\lambda_i)$ **do**
    $pos := \lambda_i$
  ifnone
    let $l = (x, \langle {}^{\lambda_1}v, \ldots, {}^{\lambda_n}v \rangle)$ **in**
      $[\![pos]\!] := (\texttt{undef}, \texttt{undef}, getValue(l))$

$(\!|\ \textbf{eval}\ {}^{\alpha}x({}^{\lambda_{i+i}}\boxed{\text{e}}, \ldots, {}^{\lambda_n}\boxed{\text{e}})\ \textbf{in}\ [{}^{\lambda_1}\boxed{\text{e}}, \ldots, {}^{\lambda_i}\boxed{\text{e}}]\ |\!) \rightarrow$
  choose $i \in [1, n]$ **with** $\neg evaluated(\lambda_i)$ **do**
    $pos := \lambda_i$
  ifnone
    let $l = (x, \langle {}^{\lambda_1}v, \ldots, {}^{\lambda_n}v \rangle)$ **in**
      $[\![pos]\!] := (\texttt{undef}, \texttt{undef}, getValue(l))$

The environment expression is expected to be a list of environment values. The order in which the elements appear in the expression corresponds to the lexical order of the equivalent nested structure of **ambient** rules. The second form of **eval** expression is useful when the identifier needs to be parameterized.

The classic global state evaluation is accessible inside **ambient** nested structure using the following expression. The **global** keyword denotes that for the identifier ${}^{\alpha}x$ all the information about environment must be discarded independently of lexical position in the specification of if.

Explicit environment evaluation

$(\!|\ \textbf{eval}\ {}^{\alpha}x\ \textbf{in global}\ |\!) \rightarrow \quad \mathsf{EvaluateId}(x, \langle\rangle)$

$(\!|\ \textbf{eval}\ {}^{\alpha}x({}^{\lambda_1}\boxed{\text{e}}, \ldots, {}^{\lambda_n}\boxed{\text{e}})\ \textbf{in global}\ |\!) \rightarrow \quad \mathsf{EvaluateId}(x, \langle \lambda_1, \ldots, \lambda_n \rangle)$

As its definition states, the **eval in global** corresponds to the classic identifier evaluation definition of CoreASM.

Chapter 7

Modularity of Data

In this chapter we discuss data representation in the ASM language and tool support for arbitrary data. Tarski structures are the foundations of abstract data representation in the ASM method and a very flexible and powerful way to manage arbitrary complex data. Any kind of data can be represented as a member of a specific domain: a set that contains elements of the same kind. The properties of such elements are defined by functions. For example if we want to represent all the possible cars one can define the universe of cars (the set containing all the possible cars) and then define the functions that describe specific properties of the single elements. Thus the function $colour$: CARS $\rightarrow$ COLOURS associate an element in the CARS set to an element in COLOURS representing its colour. The other properties are defined similarly. Sets and relations are a general enough mechanism that describes any kind of complex data.

---

universe Car
universe $Colour$

function $colour\ :\ Car \rightarrow Colour$
function $name\ :\ Car \rightarrow String$
function $manufacturer\ :\ Car \rightarrow Manufacturer$
...
$colour(c) := RED$
$manufacturer(c) := VOLKSVAGEN$
$name(c) :=$ "$Golf$"

However different approaches are possible. For example in object oriented programming, languages define data in terms of classes. A class describes the structure of every possible object in that class in terms of data and operations. Keeping the car example, in a language supporting object orientation the natural way to represent a car would be:

---

class Car{
  $colour : Colour$
  $manufacturer : Manufacturer$
  $name : String$
}

...

$c :=$ **new** $Car()$
$c.name :=$ "$Golf$"
$c.colour := RED$
$c.manufacturer := VOLKSVAGEN$

Another possibility is to represent data as algebraic data types. Algebraic data types are commonly available in functional programming languages like ML[70], Haskell[59][87], or F#[85][86]. The most common classes of algebraic data types are *product* and *sum* types. A *product type*, often called record, is an ordered tuple of types $T = A_1 \times A_2 \times \cdots \times A_n$ that represents the Cartesian product of the composing types. A *sum type*, also called tagged union, is a data structure that allows its values to belong to a set of (fixed) different types. In the following example, $Date$ is a product type of three numbers, and a $List$ of numbers is defined by two types: $Empty$ for the empty list and $Cons$ for the number at head position and the tail list.

Product and Sum types

$Date = Number \times Number \times Number$ `//Day, month and year`

$List = Empty \mid Cons$ **of** $Number,\ List$

Representing a car with product types would result in a tuple like:

record $Car = String \times Manufacturer \times Colour$
...
$c := \langle \text{“}Golf\text{”}, VOLKSVAGEN, RED \rangle$

Instead, the representation as sum types may identify the possible classes of cars:

sumtype $Car = Compact$ **of** $String \times Manufacturer \times Colour$
$\mid Suv$ **of** $String \times Manufacturer \times Colour$

$c := Compact(\text{“}Golf\text{”}, VOLKSVAGEN, RED)$

All these data representation methods have the same expressive power but depending on the context, some of them may be more suitable to describe data. Algebraic data types have been found useful in many cases, for example in [40] they are used with Event-B to model system components on abstract level with the result of simplifying the automated proofs and reduce the amount of system details during the early phase of the modelling process. In [60][35] the ASM method has been adopted to describe a formal specification of a file system for flash memory. Such specification heavily leverages freely generated algebraic data types and the authors have developed an ad hoc adaptation of algebraic data types as a plugin for CoreASM. Nevertheless, in [60], the authors write that “direct support for free data types in CoreASM would be preferable – which we have not addressed as it would require deep modifications, for example in the parser”.

The aim of this chapter is to provide a specification of algebraic data types for the ASM framework that integrates with the CoreASM environment. The context in which algebraic data types are usually found is strongly typed languages, in which their usefulness relates to the capability of compilers to statically check types and assure safe usage of algebraic data values. In spite of that, the lack of strict types in the ASM method is very useful to experiment and prototype specifications in order to better understand the problem domain. In this spirit, CoreASM implements a loosely typed language. Strict types come in handy when a specification is executed and during model verification. For this reason we are going to give first a type-less definition of algebraic data types that we will enrich afterwards with type definition and checks.

The following sections describe type-less algebraic data types (ADT), their typed version and a rule to decompose and manage ADTs. Some examples of ADT usage will be given throughout the chapter.

7.1 Algebraic data type representation

The first goal we want to achieve with algebraic data type representation is to give the specifier as little hurdle as possible in terms of notation and concepts to be known in order to begin using ADTs. At the same time we want to represent both product and sum type values. Moreover the representation should fit as close as possible the ASM framework. In order to represent product values we need ordered tuples of values. Sum type values are described by a *tag* and an element with a given signature that depends on the tag. In this scenario *lists* are the natural choice: tuples and values with a given signature can be easily represented by list values while a tag is a constant and constants are part of Tarski structures. In addition, the list concept and its representation is well known and does not require further explanation.

We establish the convention that product and sum type values are defined by lists. The sum type values are lists whose first value is a constant representing the tag. In Section 7.3 we introduce a pattern matching rule that is useful to manage lists of lists.

An example of product type value can be the representation of a car identified by its name, model and year of production. The values are tuple with signature $String \times String \times Number$. To assign two car product type to two locations it is possible to write:

Car product values

$myCar := [\text{“}Golf\text{”}, \text{“}Volkswagen\text{”}, 2013]$

$ferrari := [\text{“}FF\text{”}, \text{“}Ferrari\text{”}, 2011]$

Another example is the description of a stack of numbers to be either $Empty$ or Top of a number and the rest of the stack.

Stack tagged values

$emptyStack := [Empty]$

$someStack := [Top, 3, [Empty]]$

$anotherStack := [Top, 4, someStack]$

Notice that $anotherStack$ location is assigned to a sum type value resulting of the composition with another sum type value. In terms of equality, $[Top, 4, [Top, 3, [Empty]]]$ is considered equal to $[Top, 4, someStack]$. Product and sum values can be combined naturally. For example we can define the concept of point as a product type in $Number \times Number$ and a *circle* as a tagged value composed by a $Point$ representing its centre and a $Number$ for the diameter.

Product and Sum type composition

$myCircle := [Circle, [3, 4], 18]$

This approach does not introduce new notation, enables the use of ADTs reusing two well known concepts (lists and constants) and adheres to the principle of fast prototyping. In the first phase of the specification process the problem domain is not fully understood so a lot is continuously changing, in particular data definitions. A strict type system would require, for each change in an ADT value, to modify its definition and to align each occurrence of its values. This process is useful when the problem is fully understood and it is needed to check the uniformity of the specification, but strict types are a hindrance to the prototyping phase.

7.2 Data definition and type checking

In this section we introduce three expressions: two for the definition of product and sum types and one to type check them at runtime. The ASM framework is inherently typeless since it is based on Tarski structures that are defined by constants and predicates on them, but some form of type

definition and type checking has been implemented for example in CoreASM [37]. To talk about types we will follow the CoreASM approach since we want to be as conservative as possible with regards to the established ASM framework. In particular, we define a type checking expression that performs checks during the execution of the specification. In the CoreASM typeless language types are denoted by elements in BACKGROUND set. A background is a special universe with a static membership function. Each background contains all the elements it represents. The declaration of an ADT as product and sum types follows the syntax:

Data type declaration syntax

product $\langle NewType \rangle$ **is** $\langle Bkg \rangle [, \langle Bkg \rangle]^*$

data $\langle NewType \rangle$ **is** $\langle TagName \rangle$ **of** $\langle Bkg \rangle [, \langle Bkg \rangle]^*$ [| $\langle TagName \rangle$ **of** $\langle Bkg \rangle [, \langle Bkg \rangle]^*]^*$

The first form declares a product type named $\langle NewType \rangle$ as a product of the types denoted by the associated list of backgrounds. The second form declares a sum type $\langle NewType \rangle$ composition of a set of tagged items $\langle TagName \rangle$ **of** $\langle Bkg \rangle, \ldots, \langle Bkg \rangle$. In order to specify the semantic of ADTs we introduce the backgrounds PRODUCTBKG, SUMBKG and TAGGEDITEMBKG and some functions on elements in these backgrounds.

7.2.1 Product types

A product type is an element of PRODUCTBKG. For each product element, the functions $name_{prod}$ and $signature_{prod}$ contains the name of the element and its signature.

$$name_{prod} : \text{PRODUCTBKG} \rightarrow \text{NAME}$$
$$signature_{prod} : \text{PRODUCTBKG} \rightarrow \text{LIST(NAME)}$$

The semantic of declaring a new product type is to create a new element $e \in$ PRODUCTBKG and to define the two related function.

Product type definition

```
(| product ^α x is ^λ1 x, ..., ^λn x |) →
        if alreadyDefined(^α x) then
          Error("Already defined identifier")
        else
          choose λ ∈ {λ1, ..., λn} with ¬IsBkg(^λ x) do
            Error("Wrong type declaration")
          ifnone
            let e = new(PRODUCTBKG) in
              name_prod(e) := ^α x
              signature_prod(e) := [^λ1 x, ..., ^λn x]
              [[pos]] := (undef, undef, e)
```

Where $isBkg(b) = \exists e \in \text{ELEMENT} \wedge bkg(e) = b$.

As syntactic sugar, it would be possible to provide a variant of **data** that allow the specification of named accessors for each element in the record. The semantic would introduce function names whose definitions are the projections on the correspondent record element[1].

7.2.2 Sum types

Sum types are elements in SUMTYPEBKG. Each sum type defines a function $name_{sumtype}$ that holds the type name and $dataItems_{sumtype}$ that holds the list of tagged items composing the type.

[1]There would be name clashing problems to address or some convention on names to avoid such issue.

A tagged item is an element in TaggedItemBkg with the functions $name_{ti}$ that holds the name of the tagged item and $signature_{ti}$ containing the signature of the tagged item.

$$
\begin{aligned}
name_{ti} &: \textsc{TaggedItemBkg} \rightarrow \textsc{Name} \\
signature_{ti} &: \textsc{TaggedItemBkg} \rightarrow \textsc{List}(\textsc{Name}) \\
name_{st} &: \textsc{SumTypeBkg} \rightarrow \textsc{Name} \\
dataItem_{st} &: \textsc{TaggedItemBkg} \rightarrow \textsc{List}(\textsc{Name})
\end{aligned}
$$

The semantic of a tagged item is the evaluation of the following expression:

Data Items

$$
\begin{array}{lll}
(\!| {}^{\alpha}x \textbf{ of } {}^{\lambda_1}x, \ldots, {}^{\lambda_n}x |\!) \rightarrow & \textbf{if } pattern(parent(pos)) = \mathsf{Data} \textbf{ then} \\
& \quad \textbf{let } e = new(\textsc{TaggedItemBkg}) \textbf{ in} \\
& \quad\quad name(e) := {}^{\alpha}x \\
& \quad\quad signature(e) := \{{}^{\lambda_1}x, \ldots, {}^{\lambda_n}x\} \\
& \quad\quad [\![pos]\!] := (\mathsf{undef}, \mathsf{undef}, e) \\
& \textbf{else} \\
& \quad \mathsf{Error}(\text{“Can't use data item declaration outside data expression”})
\end{array}
$$

Notice that a tagged item expression is allowed only inside a Sum type declaration (a Data expression) Sum type semantic is so specified by:

Sum type declaration

$$
\begin{array}{lll}
(\!| \textbf{data } {}^{\alpha}x \textbf{ is } {}^{\lambda_1}\boxed{\ }, \ldots, {}^{\lambda_n}\boxed{\ } |\!) \rightarrow & \textbf{if } already Defined({}^{\alpha}x) \textbf{ then} \\
& \quad \mathsf{Error}(\text{“Already defined identifier”}) \\
& \textbf{else} \\
& \quad \textbf{choose } \lambda \in \{\lambda_1, \ldots, \lambda_n\} \textbf{ with } \neg evaluated(\lambda) \textbf{ do} \\
& \quad\quad pos := \lambda \\
& \quad \textbf{ifnone} \\
& \quad\quad \textbf{if } \exists^{\lambda} v \in \{{}^{\lambda_1}v, \ldots, {}^{\lambda_n}v\}.\neg isTagItem({}^{\lambda}v) \textbf{ then} \\
& \quad\quad\quad \mathsf{Error}(\text{“Invalid tagged item”}) \\
& \quad\quad \textbf{else} \\
& \quad\quad\quad \textbf{let } e = new(\textsc{SumTypeBkg}) \textbf{ in} \\
& \quad\quad\quad\quad dataItems(e) := [{}^{\lambda_1}v, \ldots, {}^{\lambda_n}v] \\
& \quad\quad\quad\quad name(e) := {}^{\alpha}x \\
& \quad\quad\quad\quad \textbf{forall } i \in \{{}^{\lambda_1}v, \ldots, {}^{\lambda_n}v\} \textbf{ do} \\
& \quad\quad\quad\quad\quad bkg(i) := {}^{\alpha}x \\
& \quad\quad\quad\quad [\![pos]\!] := (\mathsf{undef}, \mathsf{undef}, e)
\end{array}
$$

Where bkg is the function whose value is the background of the parameter element and $isTagItem(v) = v \in$TaggedItemBkg .

7.2.3 Type checking

In order to check the type of a value before using it, we introduce the **as** expression. This expression follows the syntax $\langle Value \rangle$ **as** $\langle TypeName \rangle$. The evaluation of **as** expressions results in the checked value when the type used for checking is compatible with the actual value type, otherwise an error is raised.

As expression

$$
\begin{array}{lll}
(\!| {}^{\alpha}\boxed{\ } \textbf{ as } {}^{\beta}x |\!) \rightarrow & pos := \alpha \\
(\!| {}^{\alpha}v \textbf{ as } {}^{\beta}x |\!) \rightarrow & [\![pos]\!] := (\mathsf{undef}, \mathsf{undef}, {}^{\alpha}v) \\
& \mathsf{CheckType}({}^{\alpha}v, {}^{\beta}x)
\end{array}
$$

To check the type we rely on the rule CheckType that raises an error if the value is incompatible with the parameter for the type.

Checking type

CheckType$(v, x) \equiv$
  if $(isADT(x) \wedge bkg(v) \neq \text{LIST}) \vee$
    $(\neg isADT(x) \wedge bkg(v) \not\sqsubseteq x) \vee$
    $(entrySet(x) = \varnothing) \vee$
    $(dataItemSet(x) = \varnothing)$ **then**
      Error("Incompatible type")
  else
    choose $e \in entrySet(x)$ **do**
      if $e \in \text{PRODUCTBKG}$ **then**
        CheckSignature$(v, signature(e))$
      else
        choose $i \in dataItemSet(x)$ **do**
          CheckSignature$(tail(v), signature(i))$

where
  $entrySet(x) = \{e \in \text{PRODUCTBKG} \cup \text{SUMTYPEBKG} | name(e) = x\}$
  $dataItemsSet(x) = \{i \in dataItem(e) | e \in \text{SUMTYPEBKG}, name(e) = x, name(i) = head(v)\}$

CheckSignature$(v, s) \equiv$
  if $(|v| \neq |s|) \vee$
    $(\exists \langle v_i, x_i \rangle : v = [v_1, \ldots, v_n] \wedge x = [x_1, \ldots, x_n] \wedge bkg(v_i) \neq x_i)$ **then**
      Error("Incompatible type")

The first step for checking types is to test if the type is an ADT with the function $isAdt$: NAME $\rightarrow$ BOOLEAN that is defined as:

$$isAdt(x) = \exists e \in \text{PRODUCTBKG} \cup \text{SUMTYPEBKG} : name(e) = x$$

The function $name$ is defined as:

$$name(x) = \begin{cases} name_{prod} & \text{if } x \in \text{PRODUCTBKG} \\ name_{st} & \text{if } x \in \text{SUMTYPEBKG} \end{cases}$$

If the type is an ADT, we proceed to check the signature of the ADT against the values of the list representing it. In this case there are two kind of failure: the signature and the list length differs[2], or there is some value whose background does not correspond to the signature. If the type is not an ADT, we just test if the background is the same or is compatible with the type. The definition of $\sqsubseteq$ is

$$a \sqsubseteq b = equals(a, b) \vee equals(\text{ELEMENT}, b)$$

This choice allows ADT definitions to have a limited form of generic type using ELEMENT as background and still successfully type check. For example we can define a *pair* as a ELEMENT×ELEMENT and check it with success against different backgrounds, like NUMBER× STRING, instead of defining pairs for each possible types. If all these condition of failure are not satisfied, the value is compatible with the denoted type.

[2]In this case we have chosen to check for length equality. Another choice would be to accept lists longer than the signature and check only the first list values.

Pair example

data *Pair* **is** *Element*, *Element*
...
$l := [3, 4]$ **as** *Pair*
$p := [$"*hello*", 34$]$ **as** *Pair*

7.3 Manipulating algebraic data types

The management of algebraic data values would became quickly overwhelming when the data grows in complexity. Manipulating lists of lists could be very tedious. Without a mechanism that deconstructs ADTs, the advantages of using them during the specification would be nullified. Most of the time, programming languages that support ADTs come with the so called *pattern matching*. Pattern matching provides a form of elimination that ease the management of algebraic data values.

A comprehensive support for ADTs in the ASM framework should not ignore pattern matching support. So we have decided to add a new transition rule for pattern matching.

The syntax of pattern matching is:

Match rule syntax

match ⟨*expr*⟩ **with** {
  | ⟨*pattern*⟩ : ⟨*rule*⟩
  | ⟨*pattern*⟩ : ⟨*rule*⟩
  ...
}

Its intuitive meaning is to check the expression against each pattern and if some pattern "matches" the expression value, the corresponding rule is enabled.

Before delving into the details of the *pattern matching* rule, we present some examples to show its intended meaning and usage.

A simple example of ADT that takes advantage of pattern matching is the representation of a *stack* structure with some operation on it to insert (push) or remove (pop) a value, or to return the value on its top (peek).

We define a stack as a sum type with two possible tagged items: *Empty* for the empty stack, and *Top* of some *Element* and the rest of the stack. So we are defining a stack that can contain all kind of values.

Stack definition

data Stack **is** Empty | Top **of** Element, Stack

The operations on stacks are specified by rules that take a stack value as parameter. These operations leverages the match rule to check if the stack is empty or not and perform the corresponding operation.

Operations on Stacks

```
Push(stack, v) ≡
  match stack with {
    | x : result := [Top, v, x]
  }

Peek(stack) ≡
  match stack with {
    | [Empty] : result := undef
    | [Top, x, _] : result := x
  }

Pop(stack) ≡
  match stack with {
    | [Empty] : result := undef
    | [Top, x, y] :  stack := y
                     result := x
```

One interesting quality of pattern matching is the ability to reason on data by cases. The stack operations are tackled decomposing the stack in its possible shapes and giving for each of them the rule definition. Pattern matching can be seen as a generalised *select-case* rule, enriched with structural match and variable bindings. Note that, in this case, the identifiers x, y inside the patterns behave as placeholders for the inner values of the matching expression. The pattern _ means that every value matches this position and no binding to such value is needed to perform the operation.

To complete the example we show a simple usage of the stack ADT. We put a stack value inside the *myStack* location and execute sequentially a *Pop* and a print of the first element in the modified stack (the number 1).

Stack usage

```
seqblock
  myStack := [Top, 3, [Top, 1, [Empty]]]
  Pop(myStack)
  print "The top value in the stack is: " + Peek(myStack)
endseqblock
```

Pattern matching can be used not only on ADT values but also on the other values such as numbers, and sets.

Another example that illustrates how abstract data types help the specification process is the definition of an algebraic expressions interpreter. We take into account only expression with additions and multiplications. An expression like $[Add, 3, 4]$ represents the addition $3+4$, similarly $[Mul, 3, 4]$ represents $3*4$. The rule Evaluate takes a numeric expression and computes the resulting value.

Expression evaluation

```
Evaluate(expr) ≡
  match expr with {
    [Const, n] : result ← n
    [Add, x, y] : result ← Evaluate(x) + Evaluate(y)
    [Mul, x, y] : result ← Evaluate(x) * Evaluate(y)
  }
```

Computing the expression $2 * 3 + 1$ can be specified directly as:

Expression evaluator usage

```
Evaluate([Add, [Mul, 2, 3], 1])
```

One final example shows how to handle polymorphic data with abstract data types. In this example we want to specify a rule that computes the area of geometric shapes. We restrict the choice of possible shapes to rectangles, circles, and triangles. A possible shape definition could be:

Shape definition

```
data SHAPE is RECT of ELEMENT, ELEMENT
            | CIRCLE of ELEMENT, ELEMENT
            | TRIANGLE of ELEMENT, ELEMENT
```

To compute the area we define a rule *ComputeArea* that takes a shape as parameter and returns the computed area.

Shape Example

```
ComputeArea(shape) ≡
  match shape with
    | [Rect, x, y]      : result := x * y
    | [Circle, r]       : result := r * r * PI
    | [Triangle, x, h]  : result := x * h/2
```

7.3.1 Pattern matching

A pattern matching rule is made of three parts: matching expression, patterns, rules. Every usual ASM expression (e.g. functions) can be placed as matching expression in the rule. Rule enabled by patterns can be every transition rule. The two most simple examples of pattern matching are the check for equality, and the identifier binding.

Simple patterns example

```
match v with {
  | 3 : R1
  | x : R2
}
```

In this example the location v is matched against the constant value 3 and an identifier x. Assuming that v has value 3, both patterns matches and the two rules R_1, R_2 are enabled. Then the rule R_2 is executed in an environment in which the identifier x is bound to the value of v.

These are simple kind of patterns, but for each match expression a great variety of them is conceivable. In order to provide maximal freedom on the possible patterns to be matched, we assume that each element domain defines its valid patterns for the expressions in it. Performing pattern matching requires knowing if the expression matches the pattern and which, if any, identifiers to bound before executing the associated rule. The *matches* function tests the compatibility of the matching expression with a pattern expression while the *bindings* function gathers the bindings between pattern identifiers and matched values.

$$matches : \text{NODE} \rightarrow \text{BOOLEAN}$$
$$bindings : \text{NODE} \rightarrow \text{SET}(\text{NAME} \times \text{ELEMENT})$$

These functions are defined as:

$$matches(n, p) = matches_{domain(n)}(n, p)$$
$$bindings(n, p) = bindings(n, p)_{domain(n)}(n, p)$$

Where *domain* is responsible for the definition of matching conditions and *bindings* rules for the corresponding value domain[3]. Now we can give the semantic for the pattern matching rule as:

$$\frac{\begin{array}{c} M = \{i | i \in \{1,\ldots,n\} \wedge matches(t, p_i)\} \\ \forall i \in M.bindings(t, p_i) = \{(x_1, v_1), \ldots, (x_h, v_h)\}, [\![R_i \frac{v_1}{x_1} \ldots \frac{v_h}{x_h}]\!]^S_\zeta = U_i \end{array}}{[\![\textbf{match } t \textbf{ with } \{p_1 : R_1 | \ldots | p_n : R_n\}]\!]^S_\zeta = \bigcup_{i \in M} U_i}$$

The related semantic for the CoreASM interpreter is the following:

```
(| match ^α[e] with |^λ1[?] → ^ρ1[r]| ... |^λn[?] → ^λn[r] |) →
        if ¬busy(pos) then
          choose i ∈ {1,...,n} with ¬matched(λ_i) do
            if matches(α, λ_i) then
              let b = bindings(α, λ_i) in
                if ∃⟨x, v1⟩, ⟨x, v2⟩ ∈ b ∧ v1 ≠ v2 then
                  Error("Multiple binding")
                else
                  busy(pos) := true
                  Bind(b)
                  toUnbind(pos) := b
                  pos := ρ_i
          ifnone
            let matchRules = {ρ ∈ ρ1,...,ρn | evaluated(ρ)} in
              [[pos]] := (undef, ⋃_{ρ∈matchRules} ^ρu, undef)
        else
          Unbind(toUnbind(pos))
          busy(pos) := false
```

Note that the semantic allows the same identifier to occur more than one time inside a pattern but the binding is considered consistent only if all the binding values are equal. The rule Bind and Unbind are responsible, respectively, for adding and removing the matching identifiers to their matched values in the environment. They are defined as:

Bind and Unbind rules

```
Bind(bindSet) ≡
  forall ⟨x, v⟩ ∈ bindSet do
    AddEnv(x, v)

Unbind(bindSet) ≡
  forall ⟨x, v⟩ ∈ bindSet do
    RemoveEnv(x)
```

The macros AddEnv and RemoveEnv are part of the CoreASM semantic, they push and remove the identifier passed as parameter to the local environment.

As syntactic sugar we introduce the **when** clause on patterns.

When clause on patterns

| ⟨*pattern*⟩**when** ⟨$expr_{when}$⟩ : ⟨*rule*⟩

This form is equivalent to

[3]In the CoreASM execution environment for the matching rules, the domain corresponds to the plugin that parses the expression

| $\langle pattern \rangle$: **if** $\langle expr_{when} \rangle$ **then** $\langle rule \rangle$

7.3.2 Matches and bindings functions examples

In this section we want to give some definition examples for the functions *matches*, and *bindings* of Section 7.3.1. We assume that every *background* (that identifies a type in CoreASM) may define its own version of *matches* and *bindings* in order to provide pattern matching expressions. For this reason we do not show the definition of such functions for every possible background but only for a relevant subset of common backgrounds. We consider patterns on values, identifiers, lists, strings, and sets. Patterns on identifiers and values represent the most basic form of matching. Patterns for lists enable the full support of ADTs, pattern matching on strings exemplifies how to use regular expressions[3] on strings, sets are another non trivial expression domain.

In the following definitions we will use the denotations:

- e for expressions,
- v for values,
- x for identifiers,
- l for lists,
- p for a generic pattern

Values and identifiers

For generic values and identifiers the matching process is defined by the following function:

$$matches_{basic}(e, \hat{e}) = \begin{cases} true & \text{if } \hat{e} = x \\ value(e) = value(\hat{e}) & \text{otherwise} \end{cases}$$

$$matches_{basic}(e, \hat{e}) = \begin{cases} \{\langle x, value(e) \rangle & \text{if } \hat{e} = x \\ \emptyset & \text{otherwise} \end{cases}$$

The basic pattern for values is an element of the value domain: if the value to be matched is equal to the pattern value, the two items match and their binding set is empty. A pattern for an identifier is always a match resulting in the binding composed by such identifier and the value being matched. With this function definitions we can write **match** rules such as

match x **with**
| $3 \rightarrow R_1$
| $[7] \rightarrow R_2$
| `North` $\rightarrow R_3$
| $y \rightarrow R_4$
| $(y) \rightarrow R_5$

In this example we match the location x against a number to check is its value is 3 or a list with 7 as single element ([7]). These two examples shows that is possible to match every usual value domain. Assuming that `North` is a constant of an enumeration, the **match** checks if the value of x is such constant. The last two rows shows how to control the bindings during the matching process. The first expression binds the value of x to the identifier y while the second evaluates the value of y and matches it against the value of x just like the first to pattern expression of this example.

Lists

For lists we introduce some patterns on their structure. The empty list pattern ($[]$) matches empty lists, the head and tail pattern ($v : l$, $x : l$) that matches the head of a list against a value or an identifier and recursively its tail.

$$
\begin{aligned}
&matches_{lists}([], []) \equiv true \\
&matches_{lists}(e : l, v : l') \equiv matches(e, v) \wedge matches_{list}(l, l') \\
&matches_{lists}(e : l, x : l') \equiv matches_{list}(l, l') \\
&matches_{lists}([], l) \equiv false \\
&matches_{lists}(l, []) \equiv false
\end{aligned}
$$

$$
\begin{aligned}
&bindings_{lists}([], []) \equiv \emptyset \\
&bindings_{lists}(e : l, v : l') \equiv bindings_{list}(l, l') \\
&bindings_{lists}(e : l, x : l') \equiv \{\langle x, value(e)\rangle\} \cup bindings_{list}(l, l') \\
&bindings_{lists}([], l) \equiv \emptyset \\
&bindings_{lists}(l, []) \equiv \emptyset
\end{aligned}
$$

With matching these pattern expression for list an example of pattern matching on list would be:

---

match $[1, 2, 3, 4]$ **with**
$\quad | \; [] \rightarrow R_1$
$\quad | \; 1 : x \rightarrow R_2$
$\quad | \; x : y \rightarrow R_3$

These patterns matches the list against the empty list, a list whose head contains 1 and whose tail is bound to the variable x, or any list with x bound to the head and y to the tail.

Sets

There are different possible patterns for sets depending on whether we are interested in the identity of their elements or not. We define the empty and singleton patterns. We do not give here a general pattern matching from a set to a set pattern, for example $\{1, 2, 3\}$ against $\{x, 2, x'\}$, since the result is not uniquely identifiable: x, x' could be both bounded to 1 and 3. One way to give such matching semantic would be to give a random bindings set as result, but here we are more interested on showing how to build *matches* and *bindings* functions and not how to define the best set patterns.

$$matches_{sets}(\{\}, \{\}) \equiv true$$
$$matches_{sets}(\{e\}, \{v\}) \equiv matches(e, v)$$
$$matches_{sets}(\{e\}, \{x\}) \equiv true$$
$$matches_{sets}(e, p) \equiv false$$

$$bindings_{sets}(\{\}, \{\}) \equiv \emptyset$$
$$bindings_{sets}(\{e\}, \{v\}) \equiv \emptyset$$
$$bindings_{sets}(\{e\}, \{x\}) \equiv \{\langle x, value(e)\rangle\}$$
$$bindings_{sets}(e, p) \equiv \emptyset$$

These pattern definition allows the **match** rule to manage sets matching in the following form:

---

match $\{1, 2, 3, 4\}$ **with**
| $\{\} \rightarrow R_1$
| $\{3\} \rightarrow R_2$
| $\{3, 1, 4, 2\} \rightarrow R_3$
| $(x) \rightarrow R_4$
| $x \rightarrow R_5$

The first pattern matches the empty set, while the second checks if the matching set is the singleton $\{3\}$. The third match equality and correspond to the basic pattern expressions. Also the last two expressions are example of basic pattern expression applied to sets. We have added them here to show how every patter definition contributes to the expressiveness of the **match** rule. In this case the two last expressions checks the variable value to be equal to the matching set and the second binds x to the set.

Strings

For strings we sketch a subset of regular expression patterns that can be used for string matching. We do not give a comprehensive definition of every regular expression operator since they are well known and here our aim is to show how to add meaningful pattern expressions on common domain elements such as strings. We denote a single characters by c and sequences of character by s. The pattern $\backslash c$ denotes that the character c must be escaped for example to match a character that also denotes an operator.

$$
\begin{aligned}
&matches_{strings}(e, \text{“}s\text{”}) \equiv match_{exact}(e, s)\\
&matches_{strings}(e, x) \equiv true\\
&matches_{strings}(e, \text{“}[s]\text{”}) \equiv match_{regex}(e, [s])\\
&matches_{strings}(e, \text{“}[\,\hat{}\,s]\text{”}) \equiv \neg match_{regex}(e, [s])\\
&matches_{strings}(e, \text{“}s*\text{”}) \equiv \neg match_{regex}(e, s*)\\
&matches_{strings}(e, e_1|e_2) \equiv match_{regex}(e, e_1) \vee matches_{regex}(e, e_2)\\
&matches_{strings}(e, e_1e_2) \equiv match_{regex}(e, e_1) \wedge matches_{regex}(e, e_2)\\
&matches_{strings}(e, \text{“}[c_1 - c_2]\text{”}) \equiv match_{regex}(e, [c_1 - c_2])\\
&matches_{strings}(e, \text{“}\backslash c\text{”}) \equiv match_{regex}(e, \text{“}\backslash c\text{”})\\
&\ldots
\end{aligned}
$$

$$
\begin{aligned}
&bindings_{strings}(e, \text{“}s\text{”}) \equiv \emptyset\\
&bindings_{strings}(e, x) \equiv \{\langle x, value(e)\rangle\}\\
&bindings_{strings}(e, \text{“}[s]\text{”}) \equiv \emptyset\\
&bindings_{strings}(e, \text{“}[\,\hat{}\,s]\text{”}) \equiv \emptyset\\
&bindings_{strings}(e, \text{“}s*\text{”}) \equiv \emptyset\\
&bindings_{strings}(e, e_1|e_2) \equiv bindings_{string}(e, e_1) \cup bindings_{string}(e, e_2)\\
&bindings_{strings}(e, e_1e_2) \equiv bindings_{string}(e, e_1) \cup bindings_{string}(e, e_2)\\
&bindings_{strings}(e, \text{“}[c_1 - c_2]\text{”}) \equiv \emptyset\\
&matches_{strings}(e, \text{“}\backslash c\text{”}) \equiv match_{regex}(e, \text{“}\backslash c\text{”})\\
&\ldots
\end{aligned}
$$

With regular expression patterns for strings we can now match expression like:

---

match s **with**
| “\b(?:(?:25[0-5]—2[0-4][0-9]—[01]?[0-9][0-9]?)\.
\b(?:(?:25[0-5]—2[0-4][0-9]—[01]?[0-9][0-9]?)\.)
\b(?:(?:25[0-5]—2[0-4][0-9]—[01]?[0-9][0-9]?)\b)” $\rightarrow R_1$
| “asm” $\rightarrow R_2$
| “\b[A-Z0-9._%+-]+@[A-Z0-9.-]+ \.[A-Z]{2,}\b” $\rightarrow R_3$

This **match** rule matches the string s against three regular expressions. The first pattern matches only strings that are IP addresses. The second pattern is a simple exact match check. The third checks if the string is a mail address. For a complete regular expressions matching, these pattern should also provide a good binding definition for the matches of the regex.

Chapter 8

Use Case

In this chapter we exemplify the usage of our modularity features for the creation of an ASM model. We model as case study the European FP7[1] project SeaClouds, a platform for the management of cloud applications.

8.1 The SeaClouds project

The SeaClouds project aims to develop an open source platform featuring seamless adaptive multi-cloud management of service-based applications. The platform consists of an application management system over IaaS (Infrastructure as a Service) and PaaS (Platform as a Service) clouds.

SeaClouds permits cloud applications developers to design, deploy, manage and configure complex applications across multiple and heterogeneous clouds

The specific objectives of SeaClouds are:

- *Orchestration* and *adaptation* of services distributed over different cloud providers. SeaClouds aims at providing the assisted design, synthesis, and simulation of service orchestrations on different cloud providers, by distributing modules of cloud-based applications over multiple heterogeneous clouds.
- *Unified application management* of services distributed over different clouds. SeaClouds is able to deploy, manage, scale and monitor services over technologically diverse cloud providers. Such operations will be performed by taking into account application requirements.
- *Monitoring* and run-time *reconfiguration* operations of services distributed over multiple heterogeneous cloud providers. Monitoring is in charge of detecting the possible need of redistributing services across cloud providers. Dynamic reconfiguration let orchestrations evolve to realise all the required changes. Reconfiguration ranges from dynamically replacing malfunctioning services to migrating them to different cloud providers

In order to achieve such objectives, the platform has been decomposed into modules. The overview of the SeaClouds architecture is shown in Figure 8.1. Six macro components are responsible for the accomplishment of all the platform functionalities. We now describe the domain and responsibilities of each of them.

Planner : It is the component in charge of analysing the user requirements, both technical and quality of service, in order to propose the application developer with a set of deployment plans. The planning process consist of the cooperation of two sub-components: the matchmaker and the optimizer. The matchmaker is in charge of filtering cloud offerings (that

[1] Framework Programme for Research and Technological Development

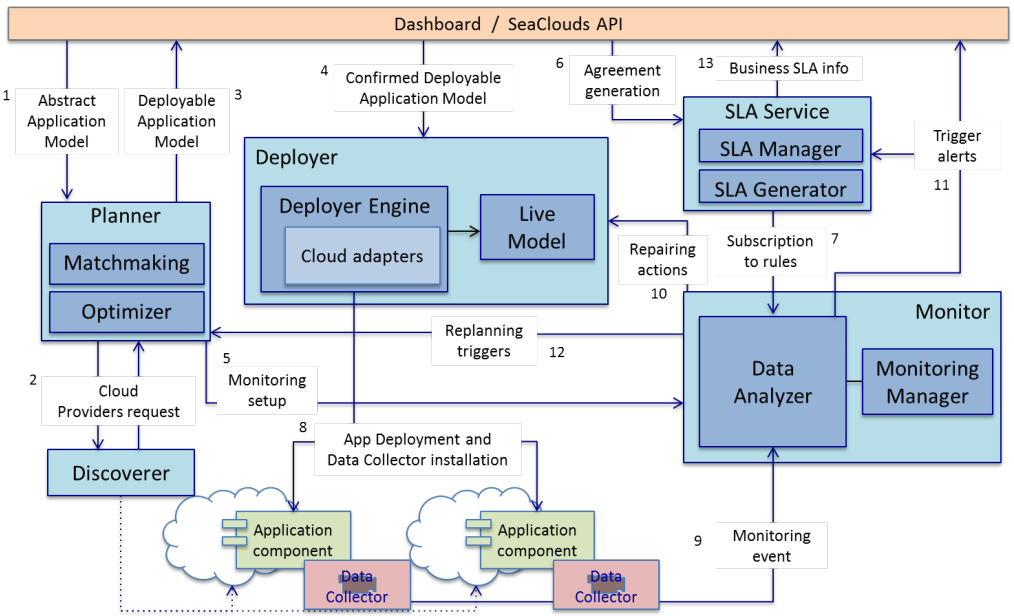


Figure 8.1: SeaClouds architecture

describe cloud provider platform capabilities) based on the user technical requirements. The optimizer takes as input the output of the matchmaker and, based on quality of service requirements, approximates the best deployment plan.

Discoverer : This component is in charge of automatically or semi-automatically look for available cloud platforms and their characteristics. A set of crawlers searches the web for these information and produce as output the cloud offerings for IaaS and PaaS providers.

Deployer : Starting from a deployment plan, the deployer is in charge of actually deploying the user application on the platform discovered and chosen during the planning phase.

SLA Service : The SLA service provides the capability to add information about Service Level Agreement and to monitor the conformance to SLA rules.

Monitor : This component is responsible for the observation of deployed applications. Based on some monitoring rule, metrics like computational load, disk usage and many other are continuously monitored to trigger, when needed, actions to repair or to signal anomalies of the running application. For example scaling policies may be triggered if the application requires better computational power.

Dashboard : The dashboard implements the orchestration of the other components providing the SeaClouds platform functionalities workflows. Moreover it implements the graphical user interface (GUI) to access the platform.

The main workflow of a typical SeaClouds use case starts from the specification of the user requirements for a cloud application that produces an Abstract Application Model (AAM). This model is the input of the planner component that starts the matchmaking process cooperating with the discoverer to get the available cloud offerings. The matchmaker produces the set of suitable offerings for each module of the application and the optimizer generates accordingly a set of abstract deployment plans (ADPs). The ADPs must be enriched with specific information (for example the cloud provider credentials) creating the final deployment application model (DAM). The DAM is the input to the deployer that eventually deploys the application.

8.2 Modelling SeaClouds with modular ASMs

In this section we show how to model a significant portion of the SeaClouds platform. We do not give a full model of all the components since they would not add more insight about the modularity constructs we introduced. The aim of this model is to provide a usage example of the new rules for a real system. Each component of the SeaClouds platform, is implemented as a web service and they communicate through REST APIs. In the model we are going to present we abstract from the communication details of REST calls by request and accept statements mechanism. We do not give a fully detailed data perspective of the REST calls parameters since they can be detailed as data refinements.

We focus on the definition of the planning phase providing the definition of a module for the Planner. It will use its sub-component (the matchmaker and the optimizer) that become also other ASM modules orchestrated by the planner module.

Planner module

```
module Planner is
  entry plan(aam) into adps
  entry replan(dam, failureInfo) into adp
  entry generateDam(adp) into dam
  do
    ambient "planner" in
      do stepwise all of
        do once
          mm := fetch from "matchmaker url"
          optimizer := fetch from "optimizer url"

        do all of
          accept plan(aam) do
            do in sequence
              request mm to match(aam)into validOfferings
              request optimizer to optimize(aam, validOfferings)into adps
            doend

          accept replan(dam, failureInfo) do
            do in sequence
              request mm to matchmake(aam)into validOfferings
              match failureInfo with
                | [failingOfferings] →
                    validOfferings := validOfferings \ failingOfferings
              request optimizer to optimize(aam, validOfferings)into adp
            doend

          accept generateDam(adp) do
            dam ← GenerateDam(adp)
        doend
      doend
```

The Planner module exposes three operations. The *plan* operation is in charge of finding a set of possible deployment plan for the user requirement described by the *aam*. The *replan* operation perform a new planning for an already deployed application that needs to be redeployed due to some failure. In order to provide the correct input for the Deployer, a DAM must be generated through the *generateDam* operation.

The module state is restricted to the "planner" environment and the two sub-components needed to perform the exposed operation are imported from URLs that we assume to be known. After the initialisation step, the planner is ready to answer to every operation request.

The first module imported from the Planner is the Matchmaker that exposes the matching service.

Mathmaker module

```
module Matchmaker is
  entry match(aam)into suitableOfferings
  do
    ambient "matchmaker" in
      do stepwise all of
        do once
          discoverer := fetch from "discoverer url"
        doend

        accept match(aam) do
          do in sequence all of
            request discoverer to getOfferingsinto offerings
            match aam with
              | [AAM, techRequirements, QoSRequirements] →
                  suitableOfferings := filter(techRequirements, offerings)
              | _ → RaiseError("InvalidAAM")
          doend
      doend
```

Similarly to the Planner, the module state is partitioned with the ambient rule and the initialisation imports the Discoverer module that will be used to fetch the available cloud offerings to be compared with the user technical requirements. Notice that the AAM is described by a sum type whose first and second parameter contain technical and quality of service requirements. The function *filter* removes the cloud offerings that do not conforms to the requirements from the available set.

Discoverer module

```
module Discoverer is
  entry getOfferings into offerings
  entry crawlOfferings into offerings
  entry getStatistics into stat
  entry deleteOffering(offer)
  do
    ambient "discoverer" in
      accept getOfferings do
        offerings ← GetOfferingsFromRepository

      accept crawlOfferings do
        do in sequence all of
          forall c ∈ Crawlers do
            ambient c in
              request c to crawlinto crawledOfferings
          forall c ∈ Crawlers do
            add crawledOfferings(c) to offerings
          UpdateStatistics
        doend

      accept getStatistics do
        stat ← GetStats

      accept deleteOffering(offer) do
        remove offer from offerings
```

The Discoverer module specifies the component that crawls for cloud offerings, that are stored into the *crawledOferings* location (parameterized accordingly to the crawler that have found them). Operations for getting statistics, and remove offerings are also provided.

the other modules can be defined similarly to the Planner component. We do not provide the modules for them since our contribution to the project was limited to the Planner and we do not have enough requirements information to generate accurate module definitions.

Following the SeaClouds platform requirements, we give the definition of the whole SeaClouds platform as a bigger module that exposes a unified API to access its functionalities. The module import all the sub-components and can be considered the global service that orchestrate all the others.

SeaCloudsPlatform

```
module SeaCloudsPlatform is
  entry generatePlans(requirements)into plans
  entry deployPlan(adp, credentials)into app
  entry getMonitoredInfo(app)into monitorInfo
  entry repairApplication(app)into plans
  do
  ambient "sc" in
    do forever  in sequence
      do once
        planner := fetch  from "planner location"
        deployer := fetch  from "deployer location"
        monitor := fetch  from "monitor location"
        sla := fetch  from "sla location"
        applications := {}

      accept generatePlans(requirements) do
        do in sequence
          aam := encodeAAM(requirements)
          request planner to plan(aam)into plans
        doend

      accept deployPlan(adp, credentials) do
        do in sequence
          request planner to generateDam(adp)into dam
          finalDam := insertCredentials(dam, credentials)
          request deployer to deploy(finalDam)into app
          add app to applications

      accept getMonitoredInfo(app) do
        request monitor to getMonitoredValues(app)into monitorInfo
      accept repairApplication(app) do
        match app with
          | [dam, [monitorderValues, failureInfo]] →
              request planner to replan(dam, failureInfo)into plans
```

Chapter 9

Conclusions

This thesis belongs to the formal methods field in software engineering. In particular, we have studied the Abstract State Machine language from the point of view of modularity features. The goal of this thesis was to improve the management of large software-intensive systems specification. We have analysed the history of the ASM method in order to understand how it has been applied and in which ways people actually write ASM specifications to tackle systems complexity.

We have found that there are at least four aspects in ASM models that are related to specifications modularization and that, if improved, enhance readability and organisation of models. We decided to improve on such modularity features to advance toward better management of ASM models. The modularity features we have studied are: control flow modularization, service oriented sub-components, state partitioning management, and uniform representation and manipulation of arbitrary data.

We have approached such modularity features at language level with the intent of providing mechanisms:

- to make large specification more manageable,
- to avoid modification of the ASM foundational semantic,
- to remain compliant with the actual tool environment (especially CoreASM).

With these objectives in mind we have proposed a set of constructs that enables modularity features into the ASM language and integrates with the CoreASM framework. This integration is achieved by the definition of the semantic of our construct as extensions of the CoreASM interpreter.

We addressed computation modularity defining the **do**-block construct that provides a uniform description of the control flow of ASM machines. This construct covers all the control flow management constructs that in time have appeared to address specific cases. The **do**-block construct manages the execution of a set of transition rules. It parameterizes conditions that allow to control the selection of the rules to be executed, when to enable execution, the strategy that defines how many and in which order the set of rules should be executed, and the termination conditions. With this construct we have contributed to unify many different control flow cases, to provide wider possibilities in the execution of set of rules and, notably to bring termination conditions back into the specification. Since, in time, many control flow rules have been proposed, the **do**-block construct is designed to be easily extensible in order to foresee future control flow constructs.

The second contribution of our thesis addresses modularity of specifications with respect to sub-systems description. We have proposed a construct to define a sub-system as an active unit of behaviour that provides a set of services. The set of services denotes the interface to interact with the sub-system. Our proposal has been inspired by ADA's tasks with rendezvous. The definition leverages the concept of agent and adds an interaction mechanism based on two transition

rules (**accept** and **request**). We have defined synchronous and asynchronous versions of such rules. This mechanism abstracts from the interaction protocol between sub-systems simplifying the specification process. Sub-systems are represented by elements in the MODULE background. To retrieve such elements we have defined two expressions: **module** and **fetch** . The **module** expression creates a new module while **fetch** provides a mechanism to retrieve modules from a given source. These two expressions allow to solve the namespace problem for modules and to reuse sub-system specifications that are retrievable as module elements. Moreover the **fetch** expression paves the way for more interesting specification environments different from the usual single file specification.

Another aspect of modularity that we have addressed is state partitioning. ASMs manage the state globally. Currently there exists few mechanisms in the ASM method for state hiding and are restricted to particular cases (like the **local** keyword). In time, people that have applied the ASM method established the practice to use location parameterization for state partitioning. This practice took various forms that have culminated into the definition of ambient ASMs. Ambient ASMs have proposed a general mechanism to describe any kind of state partition policy. Unfortunately such definition resulted unpractical. We have analysed the drawbacks of ambient ASMs and proposed an improvement of their definition. We have defined three kinds of state partition management. The state is partitioned by ambients that are the composition of environment expressions. With the **ambient set** rule, environments are managed in the same way the **amb** rule did: assigning the current environment to the identifiers inside its block. The **ambient push** rule manages the hierarchy of environments. The **ambient insert** control the composition of ambients by the management of sets of environments (preserving order independence). We analysed these three strategies and allowed their combination. To make the management of environments more fine grained we have provided a direct evaluation rule that evaluates identifier in a specific ambient parameter.

The last contribution on modularity features is related to the support for a uniform data representation. We have observed that the proper support for freely generated data types as data representation tool has been a desired feature in the ASM language that would have helped the specification process. The context in which is usual to find records and sum types is strictly typed languages, while the ASM language is typeless. So we have first defined a typeless version of them. We have proposed a convention to represent records and sum types that enables their representation with the currently available values of the ASM language. This way we do not require to learn new notation to start using abstract data types. The convention leverages the list notation and constant values. In order to allow a form of type checking we also provide abstract data type definitions and an expression (**as**) that checks if a value belongs to the expected type at specification execution time. Since we have described abstract data type values as lists of lists, their management may become soon impractical, we have proposed a pattern matching construct (**match**) that ease their manipulation. The pattern matching rule is a form of generalised select-case rule with identifiers binding. We have defined pattern matching in order to be used not only on abstract data values, but also on all the other usual values. We have assumed that each value type may define the legal patterns and their semantic. We have proposed the definitions to allow pattern matching on the most common types, in particular on lists that also represent data type values.

Examples of usage for our constructs has been provided throughout the thesis and in the use case chapter, nevertheless they should be further applied to large system specifications to establish them in the ASM language practices. Such application could highlight other aspects to be investigated and improved.

The **do**-block provides coverage for basic and Turbo ASMs control flow constructs and introduces new execution strategies (e.g. stepwise) and termination conditions. Further extensions to the construct could be investigated.

The entry-based module definition we presented generates a fixed set of services for a module. With the **accept** rule and control state ASMs (or **do**-block) the body of a module can describe when to activate and deactivate subsets of services (see the buffer example in Section 5.4). It would be interesting to allow the set of entries for a module to be dynamic for example introducing rules to add and remove entries and a way to change dynamically the body of the module. Although

this approach could deteriorate the readability of the specification it could be helpful to describe dynamic systems. Also the relation of our sub-system definition with the object oriented world should be investigated. For example entries could be related to methods but we currently provide a very limited form of entry overloading allowing input parameters to be optional.

One aspect that we have not covered in this thesis but that would improve specifications understandability is visualisation. A graphical notation for the **do**-block and **module** constructs could be provided. This notation could be used to allow graphical composition of a specification from which is possible to generate the corresponding pseudo-code model and vice versa. Another approach would be to try to adopt already available graphical notations. An example is the generation of UML diagrams (e.g. activity diagrams from a **do**-block).

Bibliography

[1] Jean-Raymond Abrial. *Modeling in Event-B: system and software engineering*. Cambridge University Press, 2010.

[2] Jean-Raymond Abrial, Jean-Raymond Abrial, and A Hoare. *The B-book: assigning programs to meanings*. Cambridge University Press, 2005.

[3] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers, Principles, Techniques*. Addison wesley, 1986.

[4] Marianna Nicolosi Asmundo and Elvinia Riccobene. "Consistent integration for sequential abstract state machines". In: *Abstract State Machines 2003*. Springer. 2003, pp. 324–340.

[5] Ralph-Johan Back and Joakim Wright. *Refinement calculus: a systematic introduction*. Springer Science & Business Media, 2012.

[6] R.J.R. Back. "On correct refinement of programs". In: *Journal of Computer and System Sciences* 23.1 (1981), pp. 49–68. ISSN: 0022-0000. DOI: http://dx.doi.org/10.1016/0022-0000(81)90005-2.

[7] Michael Balser et al. "Formal system development with KIV". In: *Fundamental approaches to software engineering*. Springer, 2000, pp. 363–366.

[8] Daniel M Berry. "Formal methods: the very idea: Some thoughts about why they work when they work". In: *Science of computer Programming* 42.1 (2002), pp. 11–27.

[9] Michel Bidoit and Peter D Mosses. *CASL User Manual: Introduction to Using the Common Algebraic Specification Language*. Vol. 1. Springer Science & Business Media, 2004.

[10] Dines Bjørner and Cliff B Jones. "he Vienna Development Method: The Meta-Language". In: *Lecture notes in computer science* 61 (1978).

[11] George Robert Blakely. "A Smalltalk Evolving Algebra and its Uses". In: (1992).

[12] E. Börger. "High-Level System Design and Analysis using Abstract State Machines". In: *Current Trends in Applied Formal Methods (FM-Trends 98)*. Ed. by D. Hutter et al. Vol. 1641. LNCS. Springer, 1999, pp. 1–43.

[13] Egon Börger. "A Logical Operational Semantics of Full Prolog: Part 1. Selection Core and Control". In: *Proceedings of the Third Workshop on Computer Science Logic*. CSL '89. Kaiserslautern, Germany: Springer-Verlag New York, Inc., 1989, pp. 36–64. ISBN: 0-387-52753-2.

[14] Egon Börger. "A Logical Operational Semantics of Full Prolog, Part II: Built-in Predicates for Database Manipulation". In: *Proceedings of the Mathematical Foundations of Computer Science 1990*. MFCS '90. London, UK, UK: Springer-Verlag, 1990, pp. 1–14. ISBN: 3-540-52953-5.

[15] Egon Börger. "A Logical Operational Semantics of Full Prolog: Part III. Built-In Predicates for Files, Terms, Arithmetic and Input-Output". English. In: *Logic from Computer Science*. Ed. by YiannisN. Moschovakis. Vol. 21. Mathematical Sciences Research Institute Publications. Springer New York, 1992, pp. 17–50. ISBN: 978-1-4612-7685-2. DOI: 10.1007/978-1-4612-2822-6_2.

[16] Egon Börger. "The origins and the development of the ASM method for high level system design and analysis". In: *Journal of Universal Computer Science* 8.1 (2002), pp. 2–74.

[17] Egon Börger, Antonio Cisternino, and Vincenzo Gervasi. "Ambient abstract state machines with applications". In: *Journal of Computer and System Sciences* 78.3 (2012), pp. 939–959.

[18] Egon Börger and Giuseppe Del Castillo. "A formal method for provably correct composition of a real-life processor out of basic components.(The APE100 Reverse Engineering Study)". In: *Engineering of Complex Computer Systems, 1995. Held jointly with 5th CSESAW, 3rd IEEE RTAW and 20th IFAC/IFIP WRTP, Proceedings., First IEEE International Conference on*. IEEE. 1995, pp. 145–148.

[19] Egon Börger and Albert Fleischmann. "Abstract state machine nets: closing the gap between business process models and their implementation". In: *Proceedings of the 7th International Conference on Subject-Oriented Business Process Management*. ACM. 2015, p. 1.

[20] Egon Börger and Uwe Glässer. "A formal specification of the PVM architecture". In: (1994).

[21] Egon Börger and Uwe Glässer. "Modelling and analysis of distributed and reactive systems using evolving algebras". In: *Evolving Algebras–Mini-Course, BRICS Technical Report BRICS-NS-95-4* (1995), pp. 128–153.

[22] Egon Börger, Yuri Gurevich, and Dean Rosenzweig. "The bakery algorithm: Yet another specification and verification". In: *Evolving Algebras* (1995), p. 116.

[23] Egon Börger and Rosario F Salamone. "CLAM specification for provably correct compilation of CLP (R) programs". In: (1995), pp. 96–130.

[24] Egon Börger and Simone Zenzaro. "Modeling for change via component-based decomposition and ASM refinement". In: *S-BPM ONE*. October, 2015, pp. 13–1.

[25] Egon Börger et al. "Towards a mathematical specification of the APE100 architecture: the APESE model". In: (1994).

[26] Luca Cardelli and Andrew D Gordon. "Ambient logic". In: *Mathematical Structures in Computer Science* (2003).

[27] Edmund M Clarke and Jeannette M Wing. "Formal methods: State of the art and future directions". In: *ACM Computing Surveys (CSUR)* 28.4 (1996), pp. 626–643.

[28] Norman H Cohen. *Ada as a second language*. McGraw-Hill Higher Education, 1995.

[29] Marcel Dausend and Alexander Raschke. "Introducing Aspect–Oriented Specification for Abstract State Machines". In: *Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Springer, 2014, pp. 174–187.

[30] Willem-Paul De Roever, Kai Engelhardt, and Karl-Heinz Buth. *Data refinement: model-oriented proof methods and their comparison*. 47. Cambridge University Press, 1998.

[31] Giuseppe Del Castillo. *The ASM Workbench: A Tool Environment for Computer-Aided Analysis and Validation of Abstract State Machine Models*. Springer, 2001.

[32] John Derrick and Eerke Boiten. *Refinement in Z and Object-Z*. Vol. 30. 4. Springer, 2001.

[33] Edsger Wybe Dijkstra, Edsger Wybe Dijkstra, and Edsger Wybe Dijkstra. *Notes on structured programming*. 1970.

[34] Eugène Dürr and Jan van Katwijk. "VDM++, a formal specification language for object-oriented designs". In: *CompEuro'92.'Computer Systems and Software Engineering', Proceedings*. IEEE. 1992, pp. 214–219.

[35] Gidon Ernst et al. "A formal model of a virtual filesystem switch". In: *arXiv preprint arXiv:1211.6187* (2012).

[36] Gidon Ernst et al. "Modular Refinement for Submachines of ASMs". In: *Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Springer, 2014, pp. 188–203.

[37] Roozbeh Farahbod. "CoreASM: An Extensible Modeling Framework & Tool Environment for High-level Design and Analysis of Distributed Systems". PhD thesis. Simon Fraser University, 2009.

[38] Roy Thomas Fielding. "Architectural styles and the design of network-based software architectures". PhD thesis. University of California, Irvine, 2000.

[39] Robert France et al. "The UML as a formal modeling notation". In: *Computer Standards & Interfaces* 19.7 (1998), pp. 325–334.

[40] Andreas Fürst et al. "Formal system modelling using abstract data types in Event-B". In: *Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Springer, 2014, pp. 222–237.

[41] Erich Gamma et al. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[42] Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. "A Metamodel-based Language and a Simulation Engine for Abstract State Machines." In: *J. UCS* 14.12 (2008), pp. 1949–1983.

[43] Yuri Gurevich. "A new thesis". In: *Abstracts, American Mathematical Society*. Vol. 6. 4. 1985, p. 317.

[44] Yuri Gurevich. "Abstract state machines: An overview of the project". In: *Foundations of Information and Knowledge Systems* (2004), pp. 6–13.

[45] Yuri Gurevich. "Evolving algebras 1993: Lipari guide". In: *Specification and validation methods* (1995), pp. 9–36.

[46] Yuri Gurevich. *Evolving algebras. A tutorial introduction. Bulletin of the EATCS,(43): 264-284*. 1991.

[47] Yuri Gurevich. "Reconsidering Turing's thesis:(toward more realistic semantics of programs)". In: (1984).

[48] Yuri Gurevich. "Sequential abstract-state machines capture sequential algorithms". In: *ACM Transactions on Computational Logic (TOCL)* 1.1 (2000), pp. 77–111.

[49] Yuri Gurevich and James K Huggins. "The semantics of the C programming language". In: *Computer Science Logic*. Springer. 1993, pp. 274–308.

[50] Yuri Gurevich and Raghu Mani. "Group membership protocol: Specification and verification". In: *E. B orger, editor, Specification and Validation Methods* (1995), pp. 295–328.

[51] Yuri Gurevich, Benjamin Rossman, and Wolfram Schulte. "Semantic essence of AsmL". In: *Formal Methods for Components and Objects*. Springer. 2004, pp. 240–259.

[52] John V Guttag and James J Horning. *Larch: languages and tools for formal specification*. Springer Science & Business Media, 2012.

[53] George T Heineman and William T Councill. "Component-based software engineering". In: *Putting the Pieces Together, Addison-Westley* (2001).

[54] CAR Hoare. "Communicating sequential processes". In: *The origin of concurrent programming*. Springer, 2002, pp. 413–443.

[55] Charles Antony Richard Hoare. "An axiomatic basis for computer programming". In: *Communications of the ACM* 12.10 (1969), pp. 576–580.

[56] Jim Huggins. "Kermit: Specification and verification". In: *E. B orger, editor, Specification and Validation Methods* (1995), pp. 247–293.

[57] Richard Hull et al. "Introducing the guard-stage-milestone approach for specifying business entity lifecycles". In: *Web services and formal methods*. Springer, 2011, pp. 1–24.

[58] Daniel Jackson. *Software Abstractions: logic, language, and analysis*. MIT press, 2012.

[59] Simon L Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[60] Maximilian Junker et al. "Simulating a Flash File System with CoreASM and Eclipse". In: *Lecture Notes in Informatics* (2011).

[61] Gregor Kiczales et al. *Aspect-oriented programming*. Springer, 1997.

[62] Leslie Lamport. "A new solution of Dijkstra's concurrent programming problem". In: *Communications of the ACM* 17.8 (1974), pp. 453–455.

[63] Leslie Lamport. "On interprocess communication". In: *Distributed computing* 1.2 (1986), pp. 86–101.

[64] Leslie Lamport. *Specifying systems: the TLA+ language and tools for hardware and software engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[65] Phillip A Laplante. "What Every Engineer Should Know about Software Engineering". In: (2007).

[66] Jean-Claude Laprie. "Dependable computing and fault-tolerance". In: *Digest of Papers FTCS-15* (1985), pp. 2–11.

[67] Luca Mearelli. "Integrating ASMs into the software development life cycle". In: *Journal of Universal Computer Science* 3.5 (1997), pp. 603–665.

[68] Luca Mearelli. "Refining an ASM specification of the production cell to C++ code". In: *Journal of Universal Computer Science* 3.5 (1997), pp. 666–688.

[69] Stephen J Mellor. *MDA distilled: principles of model-driven architecture*. Addison-Wesley Professional, 2004.

[70] Robin Milner. *The definition of standard ML: revised*. MIT press, 1997.

[71] Robin Milner, Joachim Parrow, and David Walker. "A calculus of mobile processes, i". In: *Information and computation* 100.1 (1992), pp. 1–40.

[72] C Carroll Morgan. *Programming from Specifications. International Series in Computer Science*. 1990.

[73] Joseph M. Morris. "A theoretical basis for stepwise refinement and the programming calculus". In: *Science of Computer Programming* 9.3 (1987), pp. 287–306. ISSN: 0167-6423. DOI: `http://dx.doi.org/10.1016/0167-6423(87)90011-6`.

[74] Bashar Nuseibeh and Steve Easterbrook. "Requirements engineering: a roadmap". In: *Proceedings of the Conference on the Future of Software Engineering*. ACM. 2000, pp. 35–46.

[75] Tim O'Brien et al. *Maven: The Definitive Guide is a book about Apache Maven. Number ISBN 10: 0-596-51733-5*. 2008.

[76] Brian Randell. "System structure for software fault tolerance". In: *ACM SIGPLAN Notices*. Vol. 10. 6. ACM. 1975, pp. 437–449.

[77] Dean Rosenzweig. "The WAM— definition and compiler correctness". In: *Logic Programming: Formal Methods and Practical Applications, Studies in Computer Science and Artificial Intelligence. North-Holland* (1994).

[78] James Rumbaugh et al. *Object-oriented modeling and design*. Vol. 199. 1. Prentice-hall Englewood Cliffs, 1991.

[79] Joachim Schmid. *AsmGofer*. PhD thesis, Available electronically at http://www.tydo.de/doktorarbeit.html. 1999.

[80] Joachim Schmid. *Introduction to asmgofeer*. 2001.

[81] Joachim Schmid. "Refinement and implementation techniques for abstract state machines". PhD thesis. Ulm, Universität Ulm, Diss., 2002.

[82] Ian Sommerville and Gerald Kotonya. *Requirements engineering: processes and techniques*. John Wiley & Sons, Inc., 1998.

[83] J Michael Spivey and JR Abrial. *The Z notation*. Prentice Hall Hemel Hempstead, 1992.

[84] Robert F Stärk, Joachim Schmid, and Egon Börger. *Java and the Java virtual machine: definition, verification, validation*. Springer Science & Business Media, 2012.

[85] D Syme. *F# Language Specification*. 2010.

[86] Don Syme, Adam Granicz, and Antonio Cisternino. *Expert F# 3.0*. Springer, 2012.

[87] Simon Thompson. *Haskell: the craft of functional programming*. Vol. 2. Addison-Wesley, 1999.

[88] Marc Vale. *The Evolving Algebra Semantics of COBOL-Part 1: Programs and Control*. Tech. rep. CSE-TR-162-93. EECS Dept., University of Michigan, 1993.

[89] Charles Wallace. *The semantics of the C++ programming language*. Citeseer, 1993.

[90] Niklaus Wirth. "Program development by stepwise refinement". In: *Communications of the ACM* 14.4 (1971), pp. 221–227.

[91] Jim Woodcock and Jim Davies. *Using Z: specification, refinement, and proof*. Prentice-Hall, Inc., 1996.

[92] Pamela Zave. "Classification of research efforts in requirements engineering". In: *ACM Computing Surveys (CSUR)* 29.4 (1997), pp. 315–321.

[93] Simone Zenzaro. "An ASM model for the procure to pay case study". In: *Proceedings of the 7th International Conference on Subject-Oriented Business Process Management*. ACM. 2015, p. 19.