WP4: Environment Specification Language

Deliverable D4.1

Candidate for a (Co)algebraic Interaction Computing Specification Language

Project funded by the European Commission
Information and Communication Technologies
Collaborative Project, Grant no. 318202
BIOMICS Project (Grant no. 318202)

**Contract Number:** 318202  
**Project Acronym:** BIOMICS

**Deliverable No.:** D4.1  
**Due Date:** 31/01/2015  
**Delivery Date:** 20/02/2015

**Author:** Eric Rothstein (UNIPASSAU), Daniel Schreckling (UNIPASSAU)  
**Partners contributed:** UNIPASSAU  
**Made available to:** Public

### Versioning

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Name, organisation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>31/01/2015</td>
<td>Daniel Schreckling (UNIPASSAU)</td>
</tr>
<tr>
<td>2</td>
<td>20/02/2015</td>
<td>Eric Rothstein (UNIPASSAU)</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Quality check:** Louise Newton-Clare (UH)

**Internal Reviewers:**  
Chrystopher Nehaniv (UH)  
Fariba Karimi (UH)  
Paolo Dini (UH)

---

This work is licensed under a [Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License](https://creativecommons.org/licenses/by-nc-sa/3.0/).
Abstract

The main contribution of this report is a functional behaviour-based specification language for Interaction Machines. Such language enables the behaviour-wise specification of Interaction Machines by using functions of the set \( (1 + \mathbb{D}_{beh})^V \times (1 + B)^A \) that satisfy three conditions: finiteness, connectivity and consistency.

We provide a discussion on the procedure of defining behaviour-based specification languages, and we illustrate this procedure by defining behaviour-based specification languages for Mealy and Moore machines. If \( A \) is a set of inputs and \( B \) is a set of outputs, then Mealy machines and Moore machines are behaviour-wise specified by the elements of the sets \( B^A^+ \) and \( B^{A^*} \), respectively. A behaviour-based specification for a multi-purpose Mealy machine is a subset of \( B^A^+ \), where \( B^A^+ \) is the behaviour-based specification of a universal Mealy machine.

As secondary goals, we address three main problems surrounding the behaviour-based specification of systems: representing behaviours, combining behaviours, and comparing behaviour-based specifications. We also present Abstract State Machines (ASMs) from a coalgebraic perspective, showing that they are fully compatible with the category theory framework that has been developed in Work package 2. Our interest in ASM is justified by their ability to model complex systems using a high-level language, making them a plausible framework for the specification of Interaction Machines. We show that basic ASMs can be modelled coalgebraically by describing a category for them using the polynomial functor \( ASM \). The functor \( ASM \) defines the category of \( ASM \)-systems, and allows the precise definition of the notions of \( ASM \)-bisimulations, \( ASM \)-observations, \( ASM \)-behaviours, \( ASM \)-traces, \( ASM \)-programs, and \( ASM \)-implementations.
# Table of Contents

1 Introduction .................................................. 6
   1.1 From Biology to Software - Evolutionary Computing ................. 6
   1.2 Behaviour-Based Computing ........................................ 7
       1.2.1 Abstract Systems ........................................... 7
       1.2.2 Abstract State Machines ................................... 8
       1.2.3 Coalgebraic Class Specifications ............................ 8
   1.3 Objectives .................................................. 8

2 Behaviour-Based Specification Languages .......................... 10
   2.1 Introduction .................................................. 10
   2.2 Mealy Machines Coalgebraically .................................. 11
       2.2.1 The Category of $M$-Systems ................................ 11
       2.2.2 Final $M$-systems .......................................... 12
   2.3 A Behaviour-based Specification Language for Mealy Machines .... 14
   2.4 Alternative Representations of Mealy Machine Behaviours .......... 15
   2.5 Extending Behaviour-based Specification Languages ............... 16
       2.5.1 Moore Machines Coalgebraically ............................. 16
       2.5.2 From Mealy to Moore, from Moore to Mealy ................. 16

3 Abstract State Machines, Coalgebraically ......................... 18
   3.1 Introduction .................................................. 18
   3.2 Abstract State Machines: A Primer ................................ 19
   3.3 The Category of Basic ASMs ...................................... 23
       3.3.1 ASM-Bisimulations ......................................... 24
       3.3.2 ASM-Behaviours ............................................ 24
       3.3.3 Final ASM-systems ......................................... 24
   3.4 Explanation by Example: Shabbat Elevator ........................ 25
       3.4.1 The Abstract Type $S_0$ .................................... 25
       3.4.2 Behaviour Conditioning ..................................... 26
       3.4.3 $S$-Behaviours ............................................ 27
       3.4.4 The Environment ........................................... 27
       3.4.5 The Extended Abstract Type $L$ ............................. 28

4 Candidate Specification Language for Interaction Machines ...... 30
   4.1 Introduction .................................................. 30
   4.2 $D$-Behaviours ............................................... 30
       4.2.1 A Final $D$-system ......................................... 31
       4.2.2 $D^*$-systems ............................................. 31
       4.2.3 $D^*$-behaviours ......................................... 32
   4.3 Level 1 Interaction Machines, Coalgebraically ................. 33
       4.3.1 IM*-Behaviours ........................................... 33
       4.3.2 The Interaction Conditions ................................ 34

5 Conclusion and Future Work ...................................... 36
References
Chapter 1

Introduction

One of the main objectives of the BIOMICS project is to map the behaviours of the biological cell into a “software cell” called the Interaction Machine. We are using the term “software cell” because it helps to introduce the Interaction Machine as a metaphor for the biological cell, since it shares some of its overall architecture, but in fact the aim of the project is more modest and focused on reproducing some aspects of the biochemistry of the cell. Therefore a more accurate name could be “digital biochemistry”.

Historically, there have been attempts to map biological behaviours into software. So far, evolutionary computing was successful mapping the biological behaviour of evolution into the process of software creation, but it was unable to map other biological behaviours into software itself. In this sense, the BIOMICS project is in need of a framework that allows us to describe, compose and translate the behaviours of the biological cell into Interaction Machines: a behaviour-centered framework for computing founded on category theory.

The development of this behaviour-centered framework is being carried out mainly by Work Packages 2, 3 and 4. This deliverable addresses the description and composition of system behaviours by the use of behaviour-based specification languages, and aims to provide a first specification language for Interaction Machines.

1.1 From Biology to Software - Evolutionary Computing

Evolutionary computing is a compendium of techniques that, using biologically-inspired algorithms, allows the development of systems that realise a desired behaviour or functionality\(^1\). Systems obtained using evolutionary algorithms need not have biologically-inspired behaviours, but the techniques used to develop them are inspired by phenomena in nature, like natural selection, mutations, etc. Developers that use evolutionary algorithms need not care about implementation details of the system, as functionality is the primary goal. In this sense, developers only need to focus in providing a good description of the behaviour that they want to see realised by their systems.

A description of behaviour can take many forms. For evolutionary algorithms, it is usually defined as a fitness function that classifies systems that are closer to the desired functionality with a higher grade. Those fitness functions can be interpreted as a set of input-output pairs, such that, if a good system is provided with the input, then it is able to produce an output that is very close to the one listed in the fitness function. However, it may happen that fully describing a fitness function is either impractical or impossible. For example, if we want a system that decides whether a natural number is prime or not, we can provide it a list of numbers that we already know that are prime, but given that there are an infinite number of them, we cannot list them all.

Evolutionary algorithms start from a set of randomly generated systems. During each iteration, the parameters of candidate systems are tweaked in an attempt to find fitter ones. Randomness plays an important role in evolutionary computing because it promotes the development of systems that would never be considered using traditional methods. Unfortunately, a technique based on randomness can be computationally inefficient, and might not attest the correctness of the developed system. We want to look for better alternatives to develop systems that realise particular behaviours, but without betting on randomness. Therefore, we take a look at alternative behaviour-based computing techniques.

\(^1\) We do not distinguish between the notions of behaviour and functionality.
1.2 Behaviour-Based Computing

Behaviour-based computing is a paradigm where functionality is the central pillar of software. We can consider evolutionary computing to be behaviour-based, because the developer does not care for implementation details of the systems that result from the evolutionary algorithms. The specification of systems in behaviour-based computing consists of two parts: description of functionalities, and their combination. A system that only has one functionality is a single-purpose system. Systems that combine multiple functionalities are multi-purpose systems. For example, by putting together the right components, it is possible to have the functionalities of a phone, a video player, a sound recorder, and a sound speaker in one smartphone. This concept of describing functionalities and multi-purpose systems is what we aim to capture with behaviour-based specification languages, which are discussed in Chapter 2. The behaviour-based specification languages described in this document have strong mathematical foundations in category theory, because category theory provides the tools necessary for precisely defining functionalities, their combination, and their comparison. Although many of the concepts presented in this document are thoroughly explained in Deliverable 2.1 [12], we provide the most relevant concepts when needed.

1.2.1 Abstract Systems

Abstract systems can be considered representatives of behaviour-based computing, because they abstract developers from various implementation details (like hardware, operating system, etc.), and allow them to describe the functionality of a particular system. As an example, consider the Mealy machines in Table 1 that describe the functionality of “bitwise negation”; that is, given an input sequence (say 0011), both machines produce the output sequence that corresponds to the bitwise negation of the input sequence (1100 in this case).

Machine 1 has four transitions: \( y_1 \xrightarrow{0|1} y_2, \ y_1 \xrightarrow{1|0} y_1, \ y_2 \xrightarrow{1|0} y_1, \) and \( y_2 \xrightarrow{0|1} y_2. \) Machine 2 has only two transitions: \( x \xrightarrow{0|1} x \) and \( x \xrightarrow{1|0} x. \) Although Machine 1 and Machine 2 are different in terms of the number of transitions and the number of states, they have the same functionality. In this sense, both systems in Table 1 can be considered functionally-equivalent. In fact, any system that is able to provide this functionality is considered as good as Machine 1 and Machine 2. Even the combination of both Machine 1 and Machine 2 (see Figure 1) is considered to be functionally-equivalent to Machine 1 and Machine 2.

<table>
<thead>
<tr>
<th>Machine 1</th>
<th>Machine 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="chart1.png" alt="Diagram" /></td>
<td><img src="chart2.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>

Table 1: Mealy machines describing bitwise negation.

Fig. 1: The combination of Machine 1 and Machine 2 from Table 1 is functionally equivalent to them.

---

2 The transition \( y_1 \xrightarrow{0|1} y_2 \) reads as follows: when the input 0 is provided, state \( y_1 \) makes a transition to state \( y_2 \) and a 1 is outputted.
1.2.2 Abstract State Machines

Abstract State Machines (ASM) are an engineering method that allows developers to design, analyse, document, reuse, validate, verify, implement and maintain models by using a high-level, mathematically precise general purpose language [4]. ASMs derive from evolving algebras, which are machines that provide operational semantics for algorithms [7], and they are considered a computation model that is more powerful and more universal than the standard computation models of theoretical computer science [8]. These properties make ASMs very appealing modelling tools to describe functionalities at high-levels of abstraction, thus also relevant for behaviour-based specification languages. Therefore, ASMs offer a unified framework for the description and combination of functionalities that is worth considering in the BIOMICS project.

ASMs are also worth considering because they empower the biologists with tools to describe behaviours of natural systems, even without prior knowledge\(^3\) of mathematics or computer science, unlike Petri nets or ordinary differential equations. In Chapter 3, we present ASMs from a coalgebraic perspective, aligning them with the categorical framework that was used for the description of behaviour-based specification languages introduced in Chapter 2.

1.2.3 Coalgebraic Class Specifications

Bart Jacobs proposed in [9] a coalgebraic perspective on object oriented programming. There, he presents classes and objects as functionalities defined by a functor, and restricted by behavioural assertions or behavioural equations. The following example class specification models a basic bank account (taken from [9]):

```
class spec : BankAccount

Public methods :
    bal : X \rightarrow Z
    ch : X \times Z \rightarrow X

Assertions :
    x.ch(n).bal = x.bal + n

Initially :
    new.bal = 0

end class spec
```

The behavioural assertion \(x.ch(n).bal = x.bal + n\) should be interpreted as follows: if the object \(x\) calls the method \(ch\) with parameter \(n\), then the new balance \(x.ch(n).bal\) is equal to the balance that \(x\) had \(x.bal\) plus \(n\). This condition has to be satisfied by all objects of the class \(BankAccount\). An object whose balance is 0 is called a new \(BankAccount\) object. This object is defined to be the default instantiation of an object of the class \(BankAccount\).

The work [9] by Jacobs is highly relevant for the BIOMICS project because it describes how to define functors from features of systems, and it contains formulas that allow us to derive final systems, proving most useful in Chapters 2, 3 and 4. Coalgebraic class specifications could serve the same purpose as ASMs serve for the project, but ASMs are preferred due to their high-level language.

1.3 Objectives

As previously mentioned, this deliverable addresses the description and composition of system behaviours by the use of behaviour-based specification languages, and aims to provide a first specification language for

\(^3\) A small learning phase is still needed to learn the language of ASMs, but not to understand the internal workings of ASMs. This is an unavoidable step when working with precise mathematical models.
Interaction Machines. More precisely, we provide an introduction to the functional specification of several systems; including Mealy machines, Moore Machines, Abstract State Machines and Interaction Machines. The most relevant outcome of this deliverable is a functional language for the specification of Interaction Machines, which is presented in Chapter 4.

We are aware that the functional programming paradigm is not that widespread among non-mathematicians and non-computer scientists. However, by finding inspiration in the ASM language, it will be possible to create a more friendly specification language for the description of Interaction Machines. This task is outside the scope of this deliverable, but will be addressed in Deliverable 4.2 – “Human-readable Behaviour-based Interaction Computing Specification language”, which is due on month 35.
2.1 Introduction

In the BIOMICS project we want to develop a behaviour-based specification language for Interaction Machines such that we can describe them by specifying what behaviours need to be realised. If the behaviour primitives used to describe these machines are biologically inspired, then the language would enable the description of systems that realise behaviours observed in biological systems. To achieve this goal, a deep understanding of what system behaviour is and how to represent it is required. Consequently, we start this chapter with a discussion on behaviour representation.

Traditionally, the definition of deterministic transition systems requires the specification of at least three components: the initial state, the transition function and the output function. Transition and output functions are usually defined for all inputs. Whatever function the system realised was considered the system’s behaviour. However, as we learned from the coalgebraic framework presented in Deliverable 2.1 [12], the behaviour of the system is actually determined by the image of the initial state under the semantic morphism in the system’s category. By selecting a different initial state, the system may realise a different behaviour. Thus, a deterministic system can be considered to have many behaviours, but only display one. Consequently, a non-deterministic version of the system can be considered to have the same behaviours as its deterministic counterpart, but display many behaviours at the same time.

A deterministic system with a defined initial state can be considered a representation of a single behaviour. A system without an initial state can be considered a representation of a collection of behaviours. This concept is nicely illustrated by Bart Jacobs in [9]. There, Jacobs presents classes and objects from object-oriented programming from a coalgebraic perspective. This work illustrates how classes can be seen as collections of behaviours, and how objects represent an individual instance of the behaviours in the class. In this sense, objects and classes from object-oriented programming are also representations of system behaviour: classes being collections of behaviours, and objects being representations of single behaviours.

Black boxes are representations of behaviour as well. Basically, black boxes are functions from inputs to observations. In this sense, black boxes are useful to describe systems without revealing specific details of their implementation. This has nice practical applications. For example, a company deploys a system, but wants the implementation details to remain secret. Therefore, the company promotes the system as a black box, and provides an interface to the system; that is, a defined way to give inputs to and retrieve outputs from the system. Given that the implementation details of the system are never directly available to the user, the system cannot be easily copied or reverse-engineered. Nevertheless, the user can make use of the system’s behaviour through the interface provided by the company.

Systems and their models are also representations of their own behaviour. Contrary to black boxes, systems and models can be considered “white box” representations, as not only they state what function is realised, but also how the function is realised. A system and its model may be described at different levels of abstraction. The more abstract the model, the less information about the specifics of the behaviour realised by the system. Conversely, the more concrete the model, the more detailed the representation of behaviour. Clearly, the system
Abstract State Machines (ASMs) [4] are systems that are described using a high-level but mathematically precise language, making them executable model systems. ASMs also have 1-to-1 refinements, allowing the representation of a single behaviour at many levels of abstraction. For the BIOMICS project, this is an interesting property, because it enables the description of behaviours at an appropriate level of abstraction, which can be later refined. Therefore, with ASMs, we can focus on behaviour descriptions rather than on their implementation. There is also a wide range of support tools and literature for ASMs: the basics of ASMs can be found in the book by Börger and Stärk [4], a framework for the modelling and execution of ASMs is provided by the CoreASM project [5], and the domain-specific language DKAL (Distributed Knowledge Authorisation Language) [1] for the modelling of distributed access control protocols puts ASMs to the use in the context of Computer Security. We provide more insights on ASMs in Chapter 3.

In this chapter we provide an introduction to the definition of behaviour-based specification languages. We guide the reader through the process of defining a behaviour-based specification language for Mealy machines. Fundamentally, this process consists in defining a category of Mealy machines, obtaining a final system in the category, taking the carrier set of the final system and using its elements as the language primitives; which are then combined to form specifications.

2.2 Mealy Machines Coalgebraically

Mealy machines are widely used in the specification of digital circuits. They are also used to model transducers, where the output of a system depends on the current state and a received input. Many authors, including Rutten, Jacobs and Silva have modelled Mealy machines in a coalgebraic fashion (see [16, 10, 2]). To define a behaviour-based specification language for Mealy machines, we need to be able to somehow represent Mealy machine behaviours; therefore, we also take a look at them from the perspective of category theory.

2.2.1 The Category of $\mathcal{M}$-Systems

In Deliverable 2.1 [12] there is a thorough analysis of Mealy machines from category theory. We excerpt the content that is most relevant for this deliverable. Let $A$ and $B$ be fixed sets of inputs and outputs, respectively. Consider the endofunctor $\mathcal{M} : \text{Set} \to \text{Set}$ defined, for set $X$ and function $X_1 \xrightarrow{f} X_2$, by

\[
\mathcal{M}(X) = (B \times X)^A
\]

\[
\mathcal{M}(X_1 \xrightarrow{f} X_2) = \mathcal{M}(X_1) \xrightarrow{\mathcal{M}(f)} \mathcal{M}(X_2).
\]

(1)

(2)

(We interchange the notation for functions between $X_1 \xrightarrow{f} X_2$ and $f : X_1 \to X_2$.)

The functor $\mathcal{M}$ defines the category $\text{Set}_\mathcal{M}$ of Mealy machines. The objects of $\text{Set}_\mathcal{M}$ are $\mathcal{M}$-systems (or $\mathcal{M}$-coalgebras), and the arrows of $\text{Set}_\mathcal{M}$ are $\mathcal{M}$-homomorphisms. We refer to $\text{Set}_\mathcal{M}$ as the category of $\mathcal{M}$-systems.

For an $\mathcal{M}$-system $X \xrightarrow{\alpha} \mathcal{M}(X)$, we say that the set $X$ is the carrier of $\alpha$, and the elements of $X$ are the states of $\alpha$. For $x$ in $X$ and $a$ in $A$, if

\[
\alpha(x)(a) = (b, x'),
\]

(3)

then we say that, when provided input $a$, the state $x$ outputs $b$ and transits to $x'$. Such information is also represented by $x \xrightarrow{ab} x'$, which is the notation for Mealy machines that we used in Chapter 1. Thus, $\alpha$ can be interpreted as a combination of a transition and an output function for the states in $X$.

Let $X_1 \xrightarrow{\alpha} \mathcal{M}(X_1)$ and $X_2 \xrightarrow{\beta} \mathcal{M}(X_2)$ be two $\mathcal{M}$-systems. An $\mathcal{M}$-homomorphism $X_1 \xrightarrow{h} X_2$ commutes the diagram shown in Figure 2. More precisely, for the $\mathcal{M}$-homomorphism $h$, the following equation holds:

\[
\mathcal{M}(h) \circ \alpha = \beta \circ h.
\]

(4)
(Note that not all functions $X_1 \xrightarrow{f} X_2$ are $M$-homomorphisms.)

$$
\begin{array}{c}
X_1 \\
\downarrow \alpha
\end{array}
\xrightarrow{h}

\begin{array}{c}
X_2 \\
\downarrow \beta
\end{array}

M(X_1) \xrightarrow{M(h)} M(X_2)
$$

Fig. 2: The $M$-homomorphism $h$ in the category of $M$-systems.

$M$-bisimulations are relations that capture the notion of *behaviour indistinguishability*, which makes them very useful for capturing notions of equivalence in behaviour-based specifications. An $M$-*bisimulation* between $X_1$ and $X_2$ is a relation $R \subseteq X_1 \times X_2$ satisfying, for all $(x_1, x_2)$ in $R$ and $a$ in $A$, the following conditions:

1. for all $x'_1$ in $X_1$, if $x_1 \xrightarrow{a_{b_1}} x'_1$, then there is $x'_2$ in $X_2$ such that $x_2 \xrightarrow{a_{b_2}} x'_2$ with $b_1 = b_2$ and $(x'_1, x'_2) \in R$.
2. for all $x'_2$ in $X_2$, if $x_2 \xrightarrow{a_{b_2}} x'_2$, then there is $x'_1$ in $X_1$ such that $x_1 \xrightarrow{a_{b_1}} x'_1$ with $b_1 = b_2$ and $(x'_1, x'_2) \in R$.

If the pair $(x_1, x_2)$ belongs to any $M$-bisimulation between $X_1$ and $X_2$, then we say that $x_1$ and $x_2$ are $M$-*bisimilar*; in other words, $x_1$ and $x_2$ are behaviour-wise indistinguishable. In Table 2 we present examples of $M$-bisimulations.

<table>
<thead>
<tr>
<th>M-system 1</th>
<th>M-system 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4&lt;br&gt;s→t→2</td>
</tr>
<tr>
<td>2</td>
<td>3&lt;br&gt;u→v→2</td>
</tr>
</tbody>
</table>

Table 2: The relation $\{(s, u), (t, u)\}$ is an $M$-bisimulation. The relation $\{(s, t), (t, s)\}$ is also an $M$-bisimulation.

$M$-homomorphisms are tightly related to $M$-bisimulations. Theorem 2.5 from [16] states that if $X_1 \xrightarrow{h} X_2$ is an $M$-homomorphism, then $h$ itself is an $M$-bisimulation between $X_1$ and $X_2$. In other words, if $h$ is an $M$-homomorphism, then, for all $x_1$ in $X_1$, the states $x_1$ and $h(x_1)$ are bisimilar. In this sense, proving the existence of an $M$-bisimulation relation can be done by showing the existence of an $M$-homomorphism that contains the $M$-bisimulation.

### 2.2.2 Final $M$-systems

In Deliverable 2.1 [12], the relationship between final systems and behaviour was studied. Final systems play a definitive role in the formalisation of semantics for coalgebras of polynomial functors, because their carrier sets contain *canonical representatives of system behaviour*. In this sense, final systems also play an important role in the definition of behaviour-based specification languages, because they provide the *representations of functionality* that we were looking for.

By using the formulas in [9, Lemma 6], we define the set $M_{beh}$ of $M$-*behaviours* by

$$
M_{beh} = B^{A^+}. 
$$

An $M$-behaviour is then a function from the set $A^+$ of non-empty finite sequences of elements of $A$ into $B$, mapping an input sequence to a single output. In this sense, $M$-behaviours coincide with the concept of “(sequential) machines” from [14] by Rhodes.
\( \mathcal{M} \)-behaviours also allow the definition of traces or executions\(^4\). Given an \( \mathcal{M} \)-behaviour \( \phi \) in \( \mathcal{M}_{\text{beh}} \) and the input sequence\(^5\) \( \sigma = (a_0, \ldots, a_n) \) in \( A^+ \), the trace of \( \sigma \) under \( \phi \) is defined by

\[
(\phi(\langle a_0 \rangle), \phi(\langle a_0, a_1 \rangle), \ldots, \phi(\langle a_0, \ldots, a_n \rangle)).
\]

In this sense, the trace defined in Equation 6 is the sequence of outputs observed when a system realising behaviour \( \phi \) is requested to process the input sequence \( \sigma \). (Note that, algebraically, traces are sequences of states, while coalgebraically, traces are sequences of observations.)

By using the same formulas in [9, Lemma 6], we define the \( \mathcal{M} \)-system \( \mathcal{M}_{\text{beh}} \xrightarrow{\pi_M} (B \times \mathcal{M}_{\text{beh}})^A \), for \( \phi \) in \( \mathcal{M}_{\text{beh}} \) and \( a \) in \( A \), by

\[
\pi_M(\phi)(a) = (\phi(\langle a \rangle), \phi'),
\]

with \( \phi(\langle a \rangle) \) in \( B \), and \( \phi' \) in \( \mathcal{M}_{\text{beh}} \) defined, for \( \sigma \) in \( A^+ \), by

\[
\phi'(\sigma) = \phi(a \cdot \sigma), \quad \text{where} \cdot \text{ is sequence concatenation.}
\]

The \( \mathcal{M} \)-system \( \pi_M \) is final in the category of \( \mathcal{M} \)-systems (see Deliverable 2.1 [12]). Therefore, for all \( \mathcal{M} \)-systems \( X \xrightarrow{\alpha} \mathcal{M}(X) \), there exists always a unique \( \mathcal{M} \)-homomorphism \( X \xrightarrow{\text{beh}_M} \mathcal{M}_{\text{beh}} \) called the semantic \( \mathcal{M} \)-homomorphism. Due to uniqueness, no other function commutes the diagram shown in Figure 3. As its name suggests, the semantic \( \mathcal{M} \)-homomorphism determines the semantics for the states of \( \mathcal{M} \)-systems, by pairing each state with an \( \mathcal{M} \)-behaviour.

![Fig. 3: The semantic \( \mathcal{M} \)-homomorphism \( \text{beh}_M \) exists for every \( \mathcal{M} \)-system, and it is unique.](image)

We can now describe the \( \mathcal{M} \)-system \( X \xrightarrow{\alpha} \mathcal{M}(X) \) as a collection of its functionalities. Using the semantic \( \mathcal{M} \)-homomorphism, we define the set

\[
\text{beh}_M(X) = \{\text{beh}_M(x) | x \in X\},
\]

of \( \mathcal{M} \)-behaviours realised by \( \alpha \). In particular, the final \( \mathcal{M} \)-system \( \pi_M \) realises all \( \mathcal{M} \)-behaviours, allowing it to emulate every \( \mathcal{M} \)-system. In other words, the final system \( \pi_M \) has all the functionalities of all Mealy machines in the category. In this sense, \( \pi_M \) is considered a universal system in the category of Mealy machines.

Two \( \mathcal{M} \)-systems that have the same set of realised behaviours can be considered behaviourally equivalent, even if structurally different (see Table 1 in Chapter 1). This view on systems allows us to completely abstract from implementation details and make functionality the only criterion to distinguish systems. In this sense, we are getting closer to the behaviour-based computing paradigm.

The final \( \mathcal{M} \)-system \( \pi_M \) has another interesting property: it is minimal. More precisely, no two different states in \( \mathcal{M}_{\text{beh}} \) are bisimilar. This property is known as the coinduction proof principle, and it is equivalent to the following statement (see [16], Theorem 9.3): for any \( \mathcal{M} \)-system \( X \xrightarrow{\alpha} \mathcal{M}(X) \), if \( \sim_X \) is the greatest bisimulation on \( X \), then, for \( x_1 \) and \( x_2 \) in \( X \),

\[
x_1 \sim_X x_2 \Rightarrow \text{beh}_M(x_1) = \text{beh}_M(x_2).
\]

---

\(^4\) We also do not distinguish between the notions of trace and execution.

\(^5\) Note that the notation \( \langle a_1, a_2 \rangle \) refers to the sequence where \( a_1 \) is the first element and \( a_2 \) is the second element. In this document we do not use (semi)group generators.
Consider the $\mathbb{M}$-systems 1 and 2 from Table 2. By the coinduction proof principle, we can claim that the $\mathbb{M}$-system 1 realises only one $\mathbb{M}$-behaviour, because $s$ and $t$ are bisimilar. On the other hand, the $\mathbb{M}$-system 2 realises two $\mathbb{M}$-behaviours as the states $v$ and $u$ are not bisimilar (they have different output for the input 1). Consequently, the $\mathbb{M}$-system 1 is not minimal, but the $\mathbb{M}$-system 2 is. Additionally, the $\mathbb{M}$-system 2 contains the functionality of the $\mathbb{M}$-system 1.

By using $\mathbb{M}$-behaviours, we have solved the problem of representing the functionalities of Mealy Machines. Now, by putting $\mathbb{M}$-behaviours in sets, we can solve the problem that concerns their combination. In this sense, a set of $\mathbb{M}$-behaviours describes $\mathbb{M}$-systems that realise many functionalities. Thus, we can define a behaviour-based specification language for Mealy machines.

**Example 1 (Functional Specification of $\mathbb{M}$-behaviours).** Let $A = \{0, 1\}$ and $B = \{0, 1\}$. We define two $\mathbb{M}$-behaviours $\phi_{\text{neg}}$, $\phi_{\text{comp}} : A^+ \rightarrow B$, for $\sigma$ in $A^*$ and $a$ in $A$, by

$$
\phi_{\text{neg}}(\sigma \cdot (a)) = \begin{cases} 
1, & \text{if } a = 0; \\
0, & \text{if } a = 1.
\end{cases}
$$

$$
\phi_{\text{comp}}(\sigma \cdot (a)) = \begin{cases} 
1, & \text{if } \sigma \notin \{0\}^* \text{ and } a = 0; \\
0, & \text{if } \sigma \in \{0\}^* \text{ and } a = 0; \\
0, & \text{if } \sigma \notin \{0\}^* \text{ and } a = 1; \\
1, & \text{if } \sigma \in \{0\}^* \text{ and } a = 1.
\end{cases}
$$

According to [2], the $\mathbb{M}$-behaviour $\phi_{\text{comp}}$ corresponds to the “two’s complement”. The $\mathbb{M}$-behaviour $\phi_{\text{neg}}$ corresponds to “bitwise negation”. In Figure 4, we show the translated version of the two $\mathbb{M}$-behaviours into Haskell code. The function `negation` corresponds to $\phi_{\text{neg}}$ and the function `complement` corresponds to $\phi_{\text{comp}}$. The set $\{ \phi_{\text{comp}}, \phi_{\text{neg}} \}$ specifies the dual-purpose $\mathbb{M}$-systems that are capable of calculating both the two’s complement and the bitwise negation of input sequences. Intuitively, we can combine the functionalities of systems via set union of their set of realised behaviours.

![Fig. 4: Specification of the functionalities “bitwise negation” and “two’s complement” in the functional programming language Haskell.](image)

### 2.3 A Behaviour-based Specification Language for Mealy Machines

As mentioned in the previous section, to each $\mathbb{M}$-system, we can associate the collection of the $\mathbb{M}$-behaviours that it realises. This perspective on systems naturally induces parallelism: for a system that realises many behaviours, if an input sequence is provided, then we can observe one trace per behaviour realised. For BIOMICS, this perspective on systems is useful, because we can envision the Interaction Machines as a system that contains a dynamic collection of other interdependent system functionalities.

$\mathbb{M}$-behaviours naturally fit the role of language primitives in a behaviour-based specification language for Mealy machines. Given their functional nature, $\mathbb{M}$-behaviours can be specified (and implemented) using any functional
programming language, like Haskell. The set \( \mathbb{M}_{beh} \) contains all the functionalities of \( \mathbb{M} \)-systems, thus, we can consider the set \( \mathbb{M}_{beh} \) to be a behaviour-based specification language for single-purpose Mealy machines.

To specify multi-purpose \( \mathbb{M} \)-systems, we simply use sets of \( \mathbb{M} \)-behaviours. Let \( M \) be a subset of \( \mathbb{M}_{beh} \). The set \( M \) specifies the multi-purpose \( \mathbb{M} \)-systems that realise the \( \mathbb{M} \)-behaviours contained in \( M \). In this sense, we can consider the set \( \mathcal{P}(\mathbb{M}_{beh}) \) of subsets of \( \mathbb{M}_{beh} \) to be the set of all \( \mathbb{M} \)-specifications. In other words, the set \( \mathcal{P}(\mathbb{M}_{beh}) \) is a complete functional behaviour-based specification language for \( \mathbb{M} \)-systems.

### 2.4 Alternative Representations of Mealy Machine Behaviours

A category of systems defined by a polynomial functor may have more than one final system, but they are all isomorphic (see [16]). In this sense, two final systems can provide alternative representations for the same system behaviour. This has a natural consequence for behaviour-based specification languages, as it implies that the representation of the behaviour primitives can be changed, but without altering the expressiveness of the specification language. In other words, it is possible to choose among several representation of behaviour and use the one that we find most convenient, without affecting the capabilities of the specification language. This is interesting for the BIOMICS project, because it means that we do not need to stick to the first representation of behaviour that we find for Interaction Machines, but we can keep looking until we find one that we think might be more convenient.

In this section, we illustrate this concept by changing the representation of Mealy machine behaviours from \( \mathbb{M} \)-behaviours to causal stream functions, effectively showing that there is more than one way to represent the behaviour of Mealy machines.

We first need to recall some theory. Let \( \text{nat} \) be the set of natural numbers. A stream \( \zeta \) of elements of \( A \) is an infinite sequence \( (\zeta(0), \zeta(1), \zeta(2), \ldots) \) of elements of \( A \). In other words, a stream is an element of the set \( A^{\text{nat}} \). A stream function \( \gamma : A^{\text{nat}} \to B^{\text{nat}} \) is causal if, for all \( \zeta \in A^{\text{nat}} \) and all \( n \in \text{nat} \), the output \( \gamma(\zeta)(n) \) is calculated based on the non-empty prefix \( \zeta[..n] \), defined by

\[
\zeta[..n] = (\zeta(0), \ldots, \zeta(n)).
\]

Let \( \Gamma \) be the set of all causal stream functions, which is formally defined by

\[
\Gamma = \{ \gamma : A^{\text{nat}} \to B^{\text{nat}} \mid \exists \tau : A^+ \to B. (\forall n \in \text{nat}, \zeta \in A^{\text{nat}}, \gamma(\zeta)(n) = \tau(\zeta[..n])) \}.
\]

Based on [10], Exercise 2.3.2.ii, we define the final \( \mathbb{M} \)-system \( L_{\mathbb{M}}(\Gamma) \), for \( \gamma \) in \( \Gamma \) and \( a \) in \( A \), by

\[
\rho(\gamma)(a) = (\tau(\langle a \rangle), \gamma'),
\]

with \( \gamma' \) defined, for the stream \( \zeta \) and \( n \) in \( \text{nat} \), by

\[
\gamma'(\zeta)(n) = \gamma(\langle a \rangle \cdot \zeta)(n + 1).
\]

In order to show that \( \mathbb{M} \)-behaviours can be represented as causal stream functions, we need to show that there is an isomorphism between \( \mathbb{M}_{beh} \) and \( \Gamma \). We define the function \( \text{causal} : \mathbb{M}_{beh} \to \Gamma \), for \( \phi \) in \( \mathbb{M}_{beh} \) and \( \zeta \) in \( A^{\text{nat}} \), by

\[
\text{causal}(\phi)(\zeta) = (\phi(\zeta[..0]), \phi(\zeta[..1]), \ldots, \phi(\zeta[..n]), \ldots).
\]

It is not hard to see why the function \( \text{causal} \) is a set isomorphism between \( \mathbb{M}_{beh} \) and \( \Gamma \). For two \( \mathbb{M} \)-behaviours \( \phi_1 \) and \( \phi_2 \), if \( \text{causal}(\phi_1) \) is equal to \( \text{causal}(\phi_2) \), then it must hold that \( \phi_1 \) is equal to \( \phi_2 \) by definition of \( \text{causal} \); thus \( \text{causal} \) is injective. For any causal stream function \( \gamma \), the \( \mathbb{M} \)-behaviour \( \tau \) is mapped to the causal stream function \( \gamma \) by \( \text{causal} \), and \( \tau \) always exists by definition of \( \Gamma \); thus \( \text{causal} \) is surjective. Consequently, \( \text{causal} \) is an isomorphism.

The isomorphism \( \text{causal} \) proves that there is more than one way to represent the behaviour of an \( \mathbb{M} \)-system; either by an \( \mathbb{M} \)-behaviour, or by a causal stream function. In this sense, any behaviour-based specification language can be “equipped” with isomorphisms like \( \text{causal} \), such that it is possible to specify behaviours in many different ways.
2.5 Extending Behaviour-based Specification Languages

Given that we envision the Interaction Machine to be able to realise a wide range of functionalities, we should not restrict ourselves to the behavioural specification of systems in just one category. In this section, we show how we can extend behaviour-based specification languages so that they are able to specify systems in more than one category. As a side effect, this empowers behaviour-based specification languages with tools for specification reuse. This is very convenient, because it allows us to reuse a set of core functionalities (for example, robustness and security) for the specification of systems in different categories.

To illustrate this concept of specification language extension, we define a language operator that allows the specification of Moore Machines in terms of M-behaviours; thus extending the behaviour-based specification language for Mealy machines introduced in the previous section. We start by modelling Moore machines as coalgebras.

2.5.1 Moore Machines Coalgebraically

Let $\mathcal{O} : \text{Set} \to \text{Set}$ be the functor defined, for set $X$ and function $X_1 \xrightarrow{f} X_2$, by

$$\mathcal{O}(X) = B \times X^A$$

(18)

$$\mathcal{O}(X_1 \xrightarrow{f} X_2) = \mathcal{O}(X_1) \xrightarrow{\mathcal{O}(f)} \mathcal{O}(X_2)$$

(19)

The functor $\mathcal{O}$ defines the category $\text{Set}\mathcal{O}$ of Moore machines. This time, an $\mathcal{O}$-coalgebra “unpacks” a state $x$ in $X$ by mapping it to a pair $(b, \delta_x)$, where $b$ and $\delta_x$ are the output and the transition function of $x$, respectively. Unlike M-systems, the output of $\mathcal{O}$-systems depends only on the state, not on the incoming input.

Again, by using the formulas in [9, Lemma 6], we define the set $\mathcal{O}_{beh}$ of $\mathcal{O}$-behaviours by

$$\mathcal{O}_{beh} = B^{A^*}.$$  

(20)

M-behaviours and $\mathcal{O}$-behaviours are very similar. An M-behaviour is a function from the set $A^+$ of non-empty finite sequences of elements of $A$ into $B$, while an $\mathcal{O}$-behaviour is a function from the set $A^*$ of all finite sequences of elements of $A$ into $B$. In other words, an $\mathcal{O}$-behaviour “contains” an M-behaviour, and has an additional mapping from the empty sequence $\varepsilon$ to an output in $B$.

Again, by the formulas in [9, Lemma 6], we define the $\mathcal{O}$-system $\mathcal{O}_{beh} \xrightarrow{\pi_\mathcal{O}} \mathcal{O}(\mathcal{O}_{beh})$, for $\varphi$ in $\mathcal{O}_{beh}$, by

$$\pi_\mathcal{O}(\varphi) = (\varphi(\varepsilon), \delta_\varphi);$$

(21)

with $\delta_\varphi$ defined, for $a$ in $A$ and $\sigma$ in $A^*$, by

$$\delta_\varphi(a)(\sigma) = \varphi((a) \cdot \sigma).$$

(22)

The $\mathcal{O}$-system $\pi_\mathcal{O}$ is final in the category of $\mathcal{O}$-systems [11]. Therefore, in a similar fashion to Section 2.3, we consider the set $\mathcal{P}(\mathcal{O}_{beh})$ to be the set of $\mathcal{O}$-specifications. In this sense, we also consider $\mathcal{P}(\mathcal{O}_{beh})$ to be a behaviour-based specification language for Moore machines. However, we are interested in the following question: instead of using $\mathcal{O}$-behaviours to define a behaviour-based specification language for Moore machines, is there a way to reuse M-behaviours such that we can specify all Moore machines with it? The answer is yes: by pairing M-behaviours with outputs in $B$, we are able to specify all $\mathcal{O}$-behaviours.

2.5.2 From Mealy to Moore, from Moore to Mealy

Consider the functions $\text{learn}: B \times M_{beh} \to \mathcal{O}_{beh}$, and $\text{forget}: \mathcal{O}_{beh} \to M_{beh}$ defined, for $b$ in $B$, $\phi$ in $M_{beh}$, $\varphi$ in $\mathcal{O}_{beh}$, $a$ in $A$ and $\sigma$ in $A^*$, by

$$\text{learn}(\phi)(\sigma) = \begin{cases} b, & \text{if } \sigma = \varepsilon; \\ \phi(\sigma), & \text{otherwise}. \end{cases}$$

(23)
\begin{equation}
\text{forget}(\varphi)(\sigma \cdot \langle a \rangle) = \varphi(\sigma \cdot \langle a \rangle).
\end{equation}

Essentially, the function \text{forget} removes the mapping $\varepsilon \mapsto \varphi(\varepsilon)$ from $\varphi$; thus, turning the $\mathcal{O}$-behaviour $\varphi$ into an $\mathcal{M}$-behaviour. The function \text{learn} does the opposite; it extends $\phi$ with the mapping $\varepsilon \mapsto b$, turning it into an $\mathcal{O}$-behaviour.

The function \text{learn} is surjective: given an $\mathcal{O}$-behaviour $\varphi$, the pair $(\varphi(\varepsilon), \text{forget}(\varphi))$ is mapped to $\varphi$ by \text{learn}. Additionally, \text{learn} is injective: if two pairs $(b_1, \phi_1)$ and $(b_2, \phi_2)$ are mapped to $\varphi$ by \text{learn}, then $b_1$ must be equal to $b_2$; since both are equal to $\varphi(\varepsilon)$. Also, $\phi_1$ must be equal to $\phi_2$, by definition of \text{learn}. This implies that the set $B \times M_{beh}$ is isomorphic to $O_{beh}$.

From Section 2.4, we know that any set that is isomorphic to the carrier set of a final system provides an additional representation of behaviour. In this sense, the behaviours of $\mathcal{O}$-systems can be represented as pairs of the set $B \times M_{beh}$. However, we learned something new: even though the set $O_{beh}$ is not isomorphic to $M_{beh}$, we can also use $\mathcal{O}$-behaviours to represent the behaviours of $\mathcal{M}$-systems. However, we do not recommend it because specifications are then not minimal, which is one of the reasons why we use the carrier sets of final systems. More precisely, an $\mathcal{O}$-specification may be specifying a multi-purpose $\mathcal{O}$-system, but a single-purpose $\mathcal{M}$-system. For example, for two different $\mathcal{O}$-behaviours $\varphi_1$ and $\varphi_2$, if $\text{forget}(\varphi_1)$ is equal to $\text{forget}(\varphi_2)$, then all three $\mathcal{O}$-specifications \{ $\varphi_1$ \}, \{ $\varphi_2$ \} and \{ $\varphi_1, \varphi_2$ \} refer to the same $\mathcal{M}$-specification, because \{ $\text{forget}(\varphi_1)$ \} is equal to \{ $\text{forget}(\varphi_2)$ \}.
Chapter 3

Abstract State Machines, Coalgebraically

Eric Rothstein Morris, Daniel Schreckling, Chrystopher L. Nehaniv and Egon Börger

3.1 Introduction

Abstract State Machines (ASMs) are an engineering method that allows developers to design, analyse, document, reuse, validate, verify, implement and maintain models by using a high-level, mathematically precise general purpose language [4]. ASMs have been used in different case-studies; most notably the analysis of Java and its implementation on the Java Virtual Machine [17], and the mathematical modelling of full Prolog [3]; proving a powerful modelling and analysis tool for practical purposes.

ASMs are also highly relevant for theoretical computer science. ASMs derive from evolving algebras, which are machines that provide operational semantics for algorithms [7], and they are considered a computation model that is more powerful and more universal than the standard computation models of theoretical computer science [8]. In particular, as stated by the ASM thesis [8],

“every sequential algorithm, on any level of abstraction, can be viewed as a sequential abstract state machine,”

and, most interestingly, they allow the treatment of sequential algorithms as transition systems [7].

In [2, 15, 16, 18] it is shown that different transition systems (including deterministic automata, Mealy machines, binary systems, and hyper-systems) can be described as different types of coalgebras. A coalgebra for an endofunctor $F : \text{Set} \to \text{Set}$ consists of a carrier set $X$ together with a map $\alpha : X \to F(X)$, with the functor $F$ usually being called the type of the coalgebra [18]. The coalgebraic approach to systems has an outstanding property: it allows a precise, mathematical description of the different behaviours of the systems in the category, and it enables their behavioural comparison. In other words, it is possible to compare transition systems of the same type $F$ by comparing their $F$-behaviour.

ASMs distinguish themselves from other transition systems by the way their underlying semantics are defined. For example, to define Moore and Mealy machines we require an output function and a transition function. Together, the transition and the output functions completely determine the semantics of Moore and Mealy machines. Contrary to Mealy and Moore machines, it is not necessary to define transition functions for ASMs, because their underlying semantics allow the interpretation of ASM rules. Depending on the rule provided, the ASM will update different locations, performing a state transition. In this sense, ASM rules can be considered semantically meaningful, unlike the inputs of Moore and Mealy machines. The rules of ASMs are also defined in a human-readable language, which makes them very appealing as sources of inspiration for the BIOMICS Interaction Machine specification language. In this sense, we envision Interaction Machines being systems with underlying semantics that somehow coincides with that of ASMs’.

The underlying semantics of Mealy and Moore machines is precisely modelled by the category theory framework provided in Deliverable 2.1 [12], and was summarised in Chapter 2. This precision in the definition of underlying semantics is what makes the categorical approach to systems so appealing. Given the capabilities of the category theory framework to precisely capture the underlying semantics of systems, we want to model ASMs as coalgebras, in hopes that we can transfer the knowledge we obtain from the coalgebraic version of ASMs to the Interaction Machine model. From this modelling exercise, we learn that basic ASMs can
be considered a particular subset Moore machines whose outputs are Tarksi structures (see [6]), and whose transition and output function semantics match the semantics of rule interpretation for ASMs. This hints us towards the possibility of modelling Interaction Machines as dynamic networks of automata (described in Deliverable 3.1.1 [13]), but with underlying semantics similar to that of ASMs.

We start by providing an ASM primer that contains the basic concepts and definitions for ASMs. Section 3.3 puts those concepts in a coalgebraic context, and we define the functor $\mathbb{ASM}$ that determines the category of basic ASMs. Using this coalgebraic framework, we define the concepts of $\mathbb{ASM}$-bisimilarity, $\mathbb{ASM}$-behaviours, and the universal $\mathbb{ASM}$-system $\pi_{\mathbb{ASM}}$. Next, in Section 3.4, we illustrate a coalgebraic specification for an ASM via an example: the Shabbat elevator. Finally, we extend the specification of the Shabbat elevator, and we illustrate how the coalgebraic view of ASMs is still compatible with the stepwise refinement capabilities of ASMs.

3.2 Abstract State Machines: A Primer

In order to explain the underlying semantics of ASMs, we provide an ASM primer. In this section we extend the notation and concepts originally given by Börger and Stärk in [4] to define ASMs. Some new notation is introduced because we did not find precise symbols for some concepts.

Definition 1 (Sets of Function, Relation and Rule Names). The set $F$ contains names of functions, and, in particular, contains the function names $true$, $false$, and $undef$. The set $R$ contains names of relations and, in particular, it contains the relation name $equals$ (though we simply write $=$, for convenience). Similarly, the set $C$ contains the names of rule/command, and contains, in particular, the rule name $main$.

Let $\mathbb{nat}$ be the set of natural numbers. An arity function $arity : (F \cup R \cup C) \rightarrow \mathbb{nat}$ is a function that assigns an arity to each name in $F$, $R$, and $C$. The expression $arity(g) = n$ reads “$g$ is $n$–ary”. By definition, the functions $true$, $false$, and $undef$ are nullary; as well as the rule $main$. The relation $=$ has arity 2.

Definition 2 (Signature). A signature $\Sigma$ for an ASM consists of a set $F$ of function names, a set $R$ of relation names, a set $C$ of rule names, and an arity function. We restrict the sets $F$, $R$ and $C$ to be disjoint pairwise.

Definition 3 (Superuniverse). A superuniverse $\mathcal{S}$ is a set with at least three different elements e.g. $\{1, 0, \perp\}$. These three elements are used to give interpretations to the function names $true$, $false$, and $undef$, respectively.

Definition 4 (Locations). The set $\mathbb{F}$ of function locations with superuniverse $\mathcal{S}$ is the set that, for an $n$–ary function name $f$, contains all pairs $(f, (s_1, \ldots, s_n))$ with elements $s_i$ in $\mathcal{S}$ ($1 \leq i \leq n$). Similarly, the set $\mathbb{R}$ of relation locations is the set that, for an $n$–ary relation name $r$ in $\mathcal{R}$, contains all pairs $(r, (s_1, \ldots, s_n))$ with, again, elements $s_i$ in $\mathcal{S}$ ($1 \leq i \leq n$). The set $\mathbb{L} = \mathbb{F} \cup \mathbb{R}$ is the set of all locations.

Definition 5 (Abstract State). An abstract state $\mathfrak{A}$ provides an interpretation for the locations in $\mathbb{L}$. More precisely:

- if $f$ is an $n$–ary function name, then $\mathfrak{A}$ acts as a function $f^{\mathfrak{A}} : \mathcal{S}^n \rightarrow \mathcal{S}$, mapping each function location $(f, (s_1, \ldots, s_n))$ to a value in $\mathcal{S}$;

---

6 Mainly, we could not find notation for sets of well-defined concepts; for example, sets of locations, sets of variables, sets of abstract states, etc.
From variables, functions and relations we can define the variable assignment $f^A$ that maps each relation location $(r, (s_1, \ldots, s_n))$ to a truth value.

In particular, if $f$ is a nullary function name, then $f^A$ is simply an element of $S$. Similarly, if $r$ is nullary, then $r^A$ is an element of $\{0, 1\}$.

Each abstract state is considered a valuation function for locations. Thus, for a location $l = (l, (s_1, \ldots, s_n))$, the expression $A^A(l)$ is equal to $l^A(s_1, \ldots, s_n)$.

By default we define $\text{false}^A = 0, \text{true}^A = 1, \text{undef}^A = \bot$ and $\text{bool} = \{0, 1\}$. We also define $=^A(s, s) = \text{true}^A$ for every $s$ in $S$. Additionally, if a function term $f(s_1, \ldots, s_n)$ is equal to $\bot$, then we say that $f$ is a partial function. The domain of $f$ in $A$ is the set of all tuples $(s_1, \ldots, s_n)$ in $S^n$ such that $f^A(s_1, \ldots, s_n)$ is not $\bot$.

It is possible to perform a transition from an abstract state $A$ to an abstract state $B$ by updating the value of one or more locations.

**Definition 6 (Updates).** An update for a function location $f$ is a pair $(f, s)$ with $s$ in $S$. An update for a relation location $r$ is a pair $(r, s)$ with $s$ in bool.

Given a state $A$, an update $(l, s)$ is trivial if $A^A(l) =^A s$ (using infix notation for $=^A$). Two updates $(l, s_1)$ and $(l, s_2)$ clash at $A$ if $s_1 \neq^A s_2$. A set of updates is inconsistent if it contains clashing updates.

If a set of updates $U$ is consistent at $A$, then it can be fired, yielding the state $A + U$. Such state is called the sequel of $A$ with respect to $U$. Equation 25 should hold for every location $l$ after firing a consistent update set.

$$ (A + U)(l) = \begin{cases} s, & \text{if } (l, s) \in U; \\ A(l), & \text{otherwise.} \end{cases} \tag{25} $$

**Definition 7 (Set of Variables).** A set of variables $V$ is a special set of names such that an element of $V$ cannot be mistaken by an element of any other set of names or terms; including $F$, $R$, $C$, $S$, etc. Contrary to the sets of functions and relations, variables do not have locations associated to them, but are rather given values via variable assignments.

**Definition 8 (Variable Assignment).** A variable assignment is a function $\zeta : V \to S$. Following [4], only a finite number of variables can be assigned by $\zeta$ at a time; thus, if $V$ is an infinite set, then $\zeta$ must be a partial function which partitions $V$ into two sets: the set of assigned variables (that is, the domain of $\zeta$) and the set of unassigned variables (the reserve).

We write $\zeta[v \mapsto s]$ for the variable assignment which coincides with $\zeta$ except that it assigns the element $s$ to the variable $v$; that is, for variables $v_1$ and $v_2$, we have

$$ \zeta[v_1 \mapsto s](v_2) = \begin{cases} s, & \text{if } v_1 = v_2; \\ \zeta(v_2), & \text{otherwise.} \end{cases} \tag{26} $$

From variables, functions and relations we can define the terms of the ASM.

**Definition 9 (Terms).** We recursively define the set $T$ of terms as follows:

- All variables are terms ($V \subseteq T$).
all nullary functions are terms.
- if \( f \) is an \( n \)-ary non-nullary function or relation, and \( t_1, \ldots, t_n \) are terms, then \( f(t_1, \ldots, t_n) \) is a term.

Terms are syntactic elements and have no semantic on their own. Terms are given values by abstract states by using the contents of their locations and the current variable assignment.

**Definition 10 (Term Valuation).** Given an abstract state \( \mathfrak{A} \) and a variable assignment \( \zeta \), the valuation function \( \llbracket \cdot \rrbracket_{\zeta}^\mathfrak{A} : \mathbf{T} \to \mathbf{S} \) maps a term \( t \) to an element of the superuniverse. The value \( \llbracket t \rrbracket_{\zeta}^\mathfrak{A} \) is defined by:

- if \( t \) has a variable \( v \) such that \( \zeta(v) \) is not defined, then \( \llbracket t \rrbracket_{\zeta}^\mathfrak{A} = \text{undef}^\mathfrak{A} \);
- if \( t = v \) and \( v \) is a variable, then \( \llbracket t \rrbracket_{\zeta}^\mathfrak{A} = \zeta(v) \);
- if \( t = c \) and \( c \) is a nullary function, then \( \llbracket t \rrbracket_{\zeta}^\mathfrak{A} = c^\mathfrak{A} \);
- if \( t = f(t_1, \ldots, t_n) \) with \( f \) being an \( n \)-ary function and \( t_1, \ldots, t_n \) being terms, then \( \llbracket t \rrbracket_{\zeta}^\mathfrak{A} = f^\mathfrak{A}(\llbracket t_1 \rrbracket_{\zeta}^\mathfrak{A}, \ldots, \llbracket t_n \rrbracket_{\zeta}^\mathfrak{A}) \).

If a term does not contain variables, then it is called a ground term. The result of replacing the variable \( v \) in term \( t \) everywhere by the term \( t' \) is written \( t_{\zeta}^{v \mapsto t'} \) (read “substitution by \( t' \) of \( v \) in \( t \)”).

**Definition 11 (Formula).** The formulas of an ASM are defined recursively by:

- if \( t_1 \) and \( t_2 \) are terms, then \( t_1 = t_2 \) is a formula;
- if \( \varphi \) is a formula, then \( \neg \varphi \) is a formula;
- if \( \varphi \) and \( \psi \) are formulas, then \( (\varphi \land \psi), (\varphi \lor \psi) \) and \( (\varphi \rightarrow \psi) \) are formulas;
- if \( \varphi \) is a formula and \( v \) is a variable, then \( (\forall v \varphi) \) and \( (\exists v \varphi) \) are formulas.

Formulas are used for the definition of transition rules, and their valuation depends on the valuation of terms.

**Definition 12 (Formula Valuation).** Let \( \mathfrak{A} \) be an abstract state, \( t_1 \) and \( t_2 \) be terms, \( \varphi \) and \( \psi \) be formulas, \( v \) be a variable and \( \zeta \) be a variable assignment. The valuation of formulas is defined by

\[
\begin{align*}
\llbracket t_1 = t_2 \rrbracket_{\zeta}^\mathfrak{A} &= \begin{cases} 
\text{true}, & \text{if } \llbracket t_1 \rrbracket_{\zeta}^\mathfrak{A} = \llbracket t_2 \rrbracket_{\zeta}^\mathfrak{A}; \\
\text{false}, & \text{if } \llbracket t_1 \rrbracket_{\zeta}^\mathfrak{A} \neq \llbracket t_2 \rrbracket_{\zeta}^\mathfrak{A}. 
\end{cases} \\
\llbracket \neg \varphi \rrbracket_{\zeta}^\mathfrak{A} &= \neg \llbracket \varphi \rrbracket_{\zeta}^\mathfrak{A} \\
\llbracket \varphi \land \psi \rrbracket_{\zeta}^\mathfrak{A} &= \llbracket \varphi \rrbracket_{\zeta}^\mathfrak{A} \land \llbracket \psi \rrbracket_{\zeta}^\mathfrak{A} \\
\llbracket \varphi \lor \psi \rrbracket_{\zeta}^\mathfrak{A} &= \llbracket \varphi \rrbracket_{\zeta}^\mathfrak{A} \lor \llbracket \psi \rrbracket_{\zeta}^\mathfrak{A} \\
\llbracket \varphi \rightarrow \psi \rrbracket_{\zeta}^\mathfrak{A} &= \llbracket \varphi \rrbracket_{\zeta}^\mathfrak{A} \rightarrow \llbracket \psi \rrbracket_{\zeta}^\mathfrak{A} \\
\llbracket \forall v \varphi \rrbracket_{\zeta}^\mathfrak{A} &= \forall s \in \mathbf{S} : \llbracket \varphi \rrbracket_{\zeta[s \mapsto s]}^\mathfrak{A} \\
\llbracket \exists v \varphi \rrbracket_{\zeta}^\mathfrak{A} &= \exists s \in \mathbf{S} : \llbracket \varphi \rrbracket_{\zeta[s \mapsto s]}^\mathfrak{A}
\end{align*}
\]

**Definition 13 (Rule Declaration).** A rule declaration for a rule name \( c \) of arity \( n \) is an expression

\[
c(v_1, \ldots, v_n) = P,
\]

where \( P \) is a transition rule (see Definition 14) and the variables appearing in rule \( P \) are contained in the list \( v_1, \ldots, v_n \).
The application of transition rules yields update sets.

**Definition 14 (Transition Rules).** Transition rules are syntactic constructs that allow the definition of update sets from a state $\mathfrak{A}$ and a variable assignment $\zeta$. The transitions rules are compiled in Table 3. In the case of the choose rule, the selection process is not stated explicitly, but it is assumed to exist and be deterministic. In other words, a choose rule can always be refined by using non-choose transition rules.

**Definition 15 (Variable Scope).** The rules “let $v = t \text{ in } P$,” “forall $v \text{ with } \varphi \text{ do } P$,” and “choose $v \text{ with } \varphi \text{ do } P$” make use of the variable $v$. The scope of $v$ varies:

- the scope of $v$ in “let $v = t \text{ in } P$” is $P$.
- the scope of $v$ in “forall $v \text{ with } \varphi \text{ do } P$” and “choose $v \text{ with } \varphi \text{ do } P$” is both $\varphi$ and $P$.

The occurrence of a variable $v$ is free in a transition rule if it is not in the scope of a “let $v$,” a “choose $v$,” or a “forall $v$” rule.

The application of transition rules yields update sets.

**Definition 16 (Transition Rule Application).** Given an abstract state $\mathfrak{A}$ and a variable assignment $\zeta$, the application of a transition rule $P$ to $\mathfrak{A}$ with $\zeta$ is equivalent to calculating

$$\mathfrak{A} + \text{yields}(P, \mathfrak{A}, \zeta),$$  \hspace{1cm} (28)

with the function yields defined by

<table>
<thead>
<tr>
<th>Name</th>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skip Rule</td>
<td>skip</td>
<td>Do nothing</td>
</tr>
<tr>
<td>Update rule</td>
<td>$f(t_1, \ldots, t_n) := t$</td>
<td>Update location $f([t_1]<em>\zeta, \ldots, [t_n]</em>\zeta)$ by the value of $t$.</td>
</tr>
<tr>
<td>Block rule</td>
<td>$P \text{ par } Q$</td>
<td>Execute both $P$ and $Q$ simultaneously.</td>
</tr>
<tr>
<td>Conditional rule</td>
<td>if $\varphi$ then $P$ else $Q$</td>
<td>If $[\varphi]_\zeta$ holds, then execute $P$; otherwise execute $Q$.</td>
</tr>
<tr>
<td>Let rule</td>
<td>let $v = t \text{ in } P$</td>
<td>Assign the value of $t$ to $v$, then execute $P$.</td>
</tr>
<tr>
<td>Forall rule</td>
<td>forall $v$ with $\varphi$ do $P$</td>
<td>Apply rule $P$ simultaneously to every element of the superuniverse that satisfies $\varphi$.</td>
</tr>
<tr>
<td>Choose rule</td>
<td>choose $v$ with $\varphi$ do $P$</td>
<td>Select and assign a value to $v$ that satisfies $\varphi$, then execute $P$.</td>
</tr>
<tr>
<td>Call rule</td>
<td>$c(t_1, \ldots, t_n)$</td>
<td>Execute $c(v_1, \ldots, v_n) = P$ by assigning the term $t_i$ to the free variable $v_i$, with $1 \leq i \leq n$.</td>
</tr>
</tbody>
</table>
Definition 17 (Abstract State Machine). An ASM consists of a superuniverse $S$, a set $F$ of function locations, a set $R$ of relation locations (both derived from a signature $\Sigma$ and $S$), and a set of transition rules $C$. Traditionally, an initial abstract state $I$ is also defined.

Definition 18 (Move of an ASM). An ASM can make a move (transition) from state $A$ to $B$ by rule $c(v_1, \ldots, v_n) = P$ for parameters $t_1, \ldots, t_n$, if $P$ yields a consistent update set $U$ such that $B = A + U$. State $B$ is called the next internal state after $P$. More precisely, if $A + \text{yields}(P, A, \zeta[v_1 \mapsto t_1, \ldots, v_n \mapsto t_n]) = B$, then

$$A \xleftarrow{c(t_1, \ldots, t_n)} B.$$  

Note that the original definition for the move of an ASM provided in [4, Definition 2.4.21] only addresses the main rule. With this definition, we tried to generalise this concept for every rule. This generalisation also enables us to define bisimulations for ASMs (see Section 3.3.1).

The function $\text{yields}$ plays a crucial role in the definition of the underlying semantics of ASMs, because it “charges” the transition rules with semantics. Without it, ASMs would only be able to perform moves when provided with update sets. The interpretation step that transforms ASM rules into update sets is a concept that we want to carry to the human-readable specification language that is going to be developed for Interaction Machines in Deliverable 4.2, because it adds a layer of abstraction to Interaction Machines, making them more friendly towards researchers not familiar with mathematics or computer science.

We now proceed to model ASMs coalgebraically.

3.3 The Category of Basic ASMs

Consider a superuniverse $S$, a signature $\Sigma$, the sets $F$ and $R$ respectively of function and relation locations (both derived from $\Sigma$ and $S$), and a set of transition rules $C$. We define the endofunctor $\text{ASM}$ in the category $\text{Set}$ for set $X$ by

$$\text{ASM}(X) = (S^F \times \text{bool}^R) \times X^C,$$

which defines the category of $\text{ASM}$-systems (or $\text{ASM}$-coalgebras, see Deliverable 2.1 [12, Chapter 9]). The set $S^F \times \text{bool}^R$ is the set of pairs of functions that map function locations to a value in the superuniverse $S$, and relation locations to a value in bool. In this sense, the set $S^F \times \text{bool}^R$ can be considered a set of Tarski structures. Henceforth, we interpret the abstract state $A$ as a pair of functions $(\text{loc}, \text{rel})$ such that, for $f$ in $F$, the expression $A(f)$ is equal to $\text{loc}(f)$, and for $r$ in $R$, the expression $A(r)$ is equal to $\text{rel}(r)$.

An $\text{ASM}$-system is a function $X \xrightarrow{\alpha} \text{ASM}(X)$ such that $\alpha$ “unpacks” $x$ in $X$ by mapping it to a pair in $\text{ASM}(X)$; that is,

$$\alpha(x) = (A_x, \delta_x);$$

(30)
where $A_x$ in $S^F \times \text{bool}^R$ is the abstract state of $x$ and $\delta_x$ in $X^C$ is the transition function that defines which moves the ASM can make from $A_x$ by using the transition rules in $C$. More precisely, let $\zeta$ be the variable assignment in $A_x$ and the transition rule $c(t_1, \ldots, t_n)$ represent the rule $c(v_1, \ldots, v_n)$ called with parameters $t_1, \ldots, t_n$; if $\delta_x(c(t_1, \ldots, t_n)) = x'$ and $\alpha(x') = (A_{x'}, \delta_{x'})$, then the condition

$$A_{x'} = A_x + \text{yields } (c(v_1, \ldots, v_n), A_x, \zeta[v_1 \mapsto t_1, \ldots, v_n \mapsto t_n])$$

is satisfied. Condition 31 can be considered the coalgebraic version of Definition 18, which defines what an ASM can perform.

By taking $B = S^F \times \text{bool}^R$ and $A = C$, we see that $\text{ASM}$ is a functor of the form $B \times (-)^A$; that is, a functor for Moore machines (see Section 2.5.1). Thus, we can interpret $\text{ASM}$-systems as Moore machines conditioned by Equation 31, and whose inputs are expressions that can be defined using the ASM language.

### 3.3.1 ASM-Bisimulations

Let $X_1 \xrightarrow{\alpha} \text{ASM}(X_1)$ and $X_2 \xrightarrow{\beta} \text{ASM}(X_2)$ be two $\text{ASM}$-systems. An $\text{ASM}$-bisimulation between $X_1$ and $X_2$ is a relation $R \subseteq X_1 \times X_2$ satisfying, for all $(x_1, x_2)$ in $R$ and $c(t_1, \ldots, t_n)$ in $C$,

1. for all location $l$ in $L$, it must hold that $A_{x_1}(l) = A_{x_2}(l)$;
2. if $x_1 \xrightarrow{c(t_1, \ldots, t_n)} x'_1$, then there is $x_2'$ in $X_2$ such that $x_2 \xrightarrow{c(t_1, \ldots, t_n)} x'_2$ and $(x'_1, x'_2) \in R$, and
3. if $x_2 \xrightarrow{c(t_1, \ldots, t_n)} x'_2$, then there is $x'_1$ in $X_1$ such that $x_1 \xrightarrow{c(t_1, \ldots, t_n)} x'_1$ and $(x'_1, x'_2) \in R$.

If the pair $(x_1, x_2)$ belongs to any $\text{ASM}$-bisimulation between $X_1$ and $X_2$, then we say that $x_1$ and $x_2$ are $\text{ASM}$-bisimilar and that they have the same $\text{ASM}$-behaviour. We believe that $\text{ASM}$-bisimulations are concrete instantiation of the concept of algorithm equivalence, defined by Gurevich in [8, Definition 3.2].

### 3.3.2 ASM-Behaviours

Let $x$ in $X$ and let $X \xrightarrow{\alpha} \text{ASM}(X)$ be an $\text{ASM}$-system. From Equation 30, we know that $\alpha(x) = (A_x, \delta_x)$, so $\alpha$ “unpacks” $x$ into the abstract state $A_x$ and the transition function $\delta_x$. We consider the abstract state $A_x$ to be the $\text{ASM}$-observation of the state $x$. Thus, we define the set $\text{ASM}_{\text{obs}}$ of $\text{ASM}$-observations by

$$\text{ASM}_{\text{obs}} = S^F \times \text{bool}^R.$$  

(32)

We also define the set $\text{ASM}_{\text{beh}}$ of $\text{ASM}$-behaviours is defined by

$$\text{ASM}_{\text{beh}} = (\text{ASM}_{\text{obs}})^C.$$  

(33)

An $\text{ASM}$-behaviour is a function from finite sequences of transition rules to a Tarski structure (an abstract state). An $\text{ASM}$-trace is a sequence of $\text{ASM}$-observations, each appearing whenever a transition rule is applied. Formally, for $\sigma$ in $C^*$ and an $\text{ASM}$-behaviour $\phi$ in $\text{ASM}_{\text{beh}}$, the $\text{ASM}$-trace determined by $\phi$ and $\sigma$ is defined by

$$\langle \phi(\varepsilon), \phi(\sigma[0]), \phi(\sigma[1]), \ldots, \phi(\sigma) \rangle.$$  

(34)

In this sense, an $\text{ASM}$-behaviour is a function that describes all possible $\text{ASM}$-traces of an $\text{ASM}$-system.

### 3.3.3 Final $\text{ASM}$-systems

The functor $\text{ASM}$ is polynomial (see [12, 16]); thus, by using the formulas in [9, Lemma 6], we define the final $\text{ASM}$-system $\text{ASM}_{\text{beh}} \xrightarrow{\text{ASM}} \text{ASM}(\text{ASM}_{\text{beh}})$, for an $\text{ASM}$-behaviour $\phi$, by

$$\pi_{\text{ASM}}(\phi) = (\phi(\varepsilon), \delta_{\phi}), \quad (\varepsilon \text{ is the empty sequence});$$  

(35)
with \( \phi(\varepsilon) \) being the initial abstract state of \( \phi \), and the function \( \delta_{\phi} \), an element of \((ASM_{beh})^C\), being defined for the transition rule \( e \) in \( C \) and the sequence of transition rules \( \sigma \) in \( C^* \) by
\[
\delta_{\phi}(e)(\sigma) = \phi(e \cdot \sigma).
\] (36)

Similarly to final \( \mathcal{M} \)-systems, there is a unique \( ASM \)-homomorphism from every \( ASM \)-system \( X \xrightarrow{\alpha} ASM(X) \) to \( \pi_{ASM} \) (see Section 2.2.2 and Figure 3): the semantic \( ASM \)-homomorphism. This implies that \( \pi_{ASM} \) is capable of emulating every other system in the same category of \( ASM \)-systems. In other words, the \( ASM \)-system \( \pi_{ASM} \) is a universal \( ASM \)-system.

### 3.4 Explanation by Example: Shabbat Elevator

We now illustrate how to describe an ASM coalgebraically via an example similar to the one presented in the ASM book (see [4], page 55): the correct modelling of a Shabbat elevator\(^7\).

**The Shabbat Lift.** A Shabbat lift is a special mode for an elevator to be operated during Shabbat, where buttons need not be pressed in order for the lift to operate and move. In a simplified version, the behavioural requirements of Shabbat lifts can be summarised as follows:

“A Shabbat lift moving in a particular direction stops in every floor until it reaches the last floor in its current direction. It then changes direction and repeats the process.”

The description consists of two parts: the first part is the definition of the abstract type \( S \), which defines the methods to operate a Shabbat lift, and the second part is the listing of desired behavioural conditions to be satisfied by systems that realise the behaviour of a Shabbat lift. This aligns with the methodology used by Jacobs in [9], but adapted for ASMs.

#### 3.4.1 The Abstract Type \( S \).

In this section we define the methods needed to operate a Shabbat lift. We use different notation for the different methods, depending on what they return. We do this in order to fit better with the ASM classification of functions and locations in [4]. Let \( X, Y \) and \( Z \) be sets:

- Methods with signature \( X \to Y \) are written using **this notation**. These methods are **Rules**.
- Methods with signature \( X \to bool \) are written using **this notation**. These methods are **Relations**.
- Methods with signature \( X \to Z \) are written using **this notation**. These methods are **Functions**.

The methods we defined for the abstract type \( S \) are listed in Table 4.

<table>
<thead>
<tr>
<th>Rules/Commands</th>
<th>Relations and Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textsc{changeDirection}: ( X \to X )</td>
<td>\textsc{inService}: ( X \to bool )</td>
</tr>
<tr>
<td>\textsc{emergency}: ( X \to X )</td>
<td>\textsc{stopped}: ( X \to bool )</td>
</tr>
<tr>
<td>\textsc{move}: ( X \to X )</td>
<td>\textsc{direction}: ( X \to {\uparrow, \downarrow} )</td>
</tr>
<tr>
<td>\textsc{repair}: ( X \to X )</td>
<td>\textsc{Floor}: ( X \to [1..m] )</td>
</tr>
<tr>
<td>\textsc{start}: ( X \to X )</td>
<td>\textsc{stop}: ( X \to X )</td>
</tr>
</tbody>
</table>

Table 4: Methods of the abstract type \( S \). The Shabbat lift operates between floors 1 and \( m \).

---

\( ^7 \) We write elevator and lift interchangeably.
For this example, the arity of functions, relations and rules would be 0, because the set $X$ of abstract states is always implicit. We derive a functor that defines a category of systems from the list of methods in Table 4, following the methodology described in [9]. We define the functor $S: \text{Set} \to \text{Set}$, for set $X$, by

$$
S(X) = \big([1..m] \cup \{\uparrow, \downarrow\}\big)^{\{\text{Direction, Floor}\}} \times \text{bool}^{\{\text{InService, Stopped}\}} \times X^{\{\text{ChangeDirection, Emergency, Move, Repair, Start, Stop}\}}.
$$

This functor belongs to the family of functors defined in Equation 29, in this case with superuniverse $[1..m] \cup \{\uparrow, \downarrow\}$. We now specify the behavioural conditions that we want $S$-systems to satisfy if they are to realise the behaviour of a Shabbat lifts. Given that we are talking about ASMs, this conditioning is performed by defining the transition rules.

### 3.4.2 Behaviour Conditioning

In order to make an $S$-system behave like a real Shabbat lift, we must define the transition rules in such a way that the values of locations are adequately changed whenever they are executed. For the initial conditions of a Shabbat lift, we only require the location InService to be true. Other locations can have any valid value (see Equation 38).

```
Floor \in [1..m] \\
Direction \in \{\uparrow, \downarrow\} \\
\text{InService} = \text{true} \\
\text{Stopped} \in \text{bool}
```

Equation 38 can be considered to be defining a set of Initial abstract states. We now define the transition rules such that locations are updated accordingly when the ASM executes them.

**Emergency.** If the EMERGENCY rule is received, the lift is deemed out of service and it stops.

```
\text{EMERGENCY} = \\
\text{Stopped} := \text{true} \\
\text{InService} := \text{false}.
```

(The vertical positioning of the the updates rules Stopped := true and InService := false implies that they executed in parallel.)

**Repair.** The REPAIR rule cancels the out-of-service status.

```
\text{REPAIR} = \text{InService} := \text{true}.
```

**Start.** The START rule allows the lift to move, but only if it is in service.

```
\text{START} = \text{if InService then Stopped} := \text{false}.
```

**Stop.** The STOP rule stops the lift.

```
\text{STOP} = \text{Stopped} := \text{true}.
```

**Change Direction.** Whenever the CHANGEDIRECTION rule is executed, the lift changes its current direction.

```
\text{CHANGEDIRECTION} = \\
\text{if Direction} = \uparrow \text{ then Direction} := \downarrow. \\
\text{if Direction} = \downarrow \text{ then Direction} := \uparrow.
```
Move. If the lift executes the MOVE rule, it should move one floor in its current direction.

\[
\text{MOVE} =
\begin{align*}
&\text{if } \text{Direction } = \uparrow \text{ then } \text{Floor } := \text{Floor } + 1 \\
&\text{if } \text{Direction } = \downarrow \text{ then } \text{Floor } := \text{Floor } - 1.
\end{align*}
\] (44)

We now define the set of all possible different Shabbat lift behaviours.

3.4.3 $S$-Behaviours

Given that the arity of all functions and relations for the Shabbat lift is 0 (see Section 3.4), the sets of function locations is

\[
F = \{\text{Direction}, \text{Floor}, \text{FirstFloor}, \text{LastFloor}\};
\] (45)

the set of relation locations is

\[
R = \{\text{INService, Stopped}\};
\] (46)

and the set of transition rules is

\[
C = \{\text{CHANGEDirection, EMERGENCY, MOVE, REPAIR, START, STOP}\}.
\] (47)

Now, from Section 3.3, we know that the set of $S$-observations (or abstract states of $S$-systems) is

\[
S_{obs} = ([1..m] \cup \{\uparrow, \downarrow\})^F \times \text{bool}^R,
\] (48)

and that the set of $S$-behaviours is

\[
S_{beh} = (S_{obs})^C;
\] (49)

which defines all possible behaviours that $S$-systems can realise. Remember that if a $S$-system satisfies Condition 31, then it can be considered viable model for a Shabbat Lift. Note that a $S$-behaviours describes an ASM that models a Shabbat lift as a function.

3.4.4 The Environment

The correctness condition of a Shabbat lift is a liveness property that states

“all floors are serviced eventually, with floors given equal priority.”

Note that the correctness condition does not provide details on how a Shabbat lift should be implemented, as many possible implementations satisfy it. For example, an $S$-system that stops on every floor when going up and stops on every floor when going down satisfies the correctness condition. However, an $S$-system that stops on even floors when going up, and on odd floors when going down also satisfies the correctness condition. Even an $S$-system that randomly selects floors to visit without repeating satisfies the correctness condition. This inability of the correctness condition to determine implementation details cannot be avoided, because the correctness condition is a property of single traces. More precisely, given an $\sigma$ in $C^*$ and $\phi$ in $S_{beh}$, the $S$-trace determined by $\phi$ and $\sigma$, which is defined by

\[
\langle \phi(\varepsilon), \phi(\sigma[0]), \phi(\sigma[1]), \ldots, \phi(\sigma) \rangle
\] (50)

may or may not satisfy the correctness condition. It is then responsibility of the environment to provide an input sequence that allows the satisfaction of the correctness condition. Thus, by fixing $\sigma$ and $\phi$ such that the correctness condition is satisfied, we can consider to have defined an implementation for the Shabbat lift. We
machines. For this example, it is necessary to define the rules similar to the procedure carried out in Section 2.5. We can define all possible characteristics of the ASM method.

More precisely, we show how to extend an existing ASM-system so it can handle more inputs and/or produce more outputs. This is brings us closer to an incremental, stepwise refinement of systems, which is one of the advantages of the ASM method.

In this section we illustrate that ASM-systems can be extended to be able to display more complex behaviours. More precisely, we show how to extend an existing ASM-behaviour so it can handle more inputs and/or produce more outputs. This is brings us closer to an incremental, stepwise refinement of systems, which is one of the advantages of the ASM method.

Although behaviours are functions from finite sequences to observations, they can be extended for streams. We refer to the streams of transition rules as ASM-program. For example, if the Shabbat lift starts on floor 1, has direction ↑, is stopped, and satisfies the initial conditions (see Equation 38), then the ASM-program

\[ \sigma_1 = \left( (\text{START} \cdot \text{MOVE} \cdot \text{STOP})^{m-1} \cdot \text{CHANGE DIRECTION} \right)^\omega \]  

allows the Shabbat lift to satisfy the correctness condition. Note that this ASM-program is the infinite repetition of the finite sequence

\[ (\text{START} \cdot \text{MOVE} \cdot \text{STOP})^{m-1} \cdot \text{CHANGE DIRECTION}, \]  

which is a sequence that produces a trace that satisfies the correctness condition on its own. The ASM-program

\[ \sigma_2 = \left( (\text{REPAIR} \cdot \text{START} \cdot \text{MOVE} \cdot \text{EMERGENCY})^{m-1} \cdot \text{CHANGE DIRECTION} \right)^\omega \]  

also produces traces that satisfy the correctness condition, although ASM-programs \( \sigma_1 \) and \( \sigma_2 \) produce different traces (\( \sigma \) never updates value of the location INSERVICE, unlike \( \sigma' \)).

We can now formalise the notion of an ASM-implementation, which is essentially a pair \((\phi, \sigma)\), where \( \phi \) is an ASM-behaviour and \( \sigma \) is an ASM-program.

### 3.4.5 The Extended Abstract Type \( \mathbb{L} \)

In this section we illustrate that ASM-systems can be extended to be able to display more complex behaviours. More precisely, we show how to extend an existing ASM-behaviour so it can handle more inputs and/or produce more outputs. This is brings us closer to an incremental, stepwise refinement of systems, which is one of the advantages of the ASM method.

Consider the type \( \mathbb{L} \), and extension to the type \( \mathbb{S} \) shown in Table 5. Note that the arity of \textsc{RequestFloor} and \textsc{FloorRequested} is 1, and that the arity of \textsc{RequestLift} and \textsc{LiftRequested} is 2. The functor \( \mathbb{L} \) is defined, for set \( X \), by

\[
\mathbb{L}(X) = \left( \left( [1..m] \cup \{\uparrow, \downarrow\} \right)^F \times \text{bool}^R \times \{\text{FloorRequested}\} \times [1..m] \times \{\text{LiftRequested}\} \times [1..m] \times \{\uparrow, \downarrow\} \right) \times X^{\{\text{RequestFloor}\} \times [1..m] \cup \{\text{RequestLift}\} \times [1..m] \times \{\uparrow, \downarrow\}}
\]  

We can define all possible \( \mathbb{L} \)-behaviours by extending \( \mathbb{S} \)-behaviours with additional information. This is similar to the procedure carried out in Section 2.5 where we defined Moore machines by extending Mealy machines. For this example, it is necessary to define the rules \textsc{RequestFloor} and \textsc{RequestLift}, as well
as review whether the previously defined rules of the Shabbat lift affect the locations $FLOOR_{REQUESTED}(f)$ and $LIFT_{REQUESTED}(f, d)$, for $f$ in $[1..m]$ and $d$ in $\{\uparrow, \downarrow\}$.

If the lift receives the $\text{REQUESTFLOOR}$ rule with parameter $f$, then a request for floor $f$ is acknowledged.

$$\text{REQUESTFLOOR}(f) = FLOOR_{REQUESTED}(f) := \text{true}. \quad (55)$$

Similarly, if the $\text{REQUESTLIFT}$ rule is received with parameter $(f, d)$, then a request for a lift going in direction $d$ is generated on floor $f$.

$$\text{REQUESTFLOOR}(f, d) = LIFT_{REQUESTED}(f, d) := \text{true}. \quad (56)$$

However if the lift is required to $\text{STOP}$, the existing requests for that floor should be cancelled; more precisely:

$$\text{STOP} =$$

$$\text{STOPPED} := \text{true}$$

$$\text{let } f = \text{Floor}, d = \text{Direction} \text{ in }$$

$$FLOOR_{REQUESTED}(f) := \text{false}$$

$$LIFT_{REQUESTED}(f, d) := \text{false}$$

We can also define $L$-observations and $L$-behaviours. The sets

$$R_L = \{FLOOR_{REQUESTED}\} \times [1..m] \cup \{LIFT_{REQUESTED}\} \times [1..m] \times \{\uparrow, \downarrow\}, \quad (58)$$

$$C_L = \{\text{REQUESTFLOOR}\} \times [1..m] \cup \{\text{REQUESTLIFT}\} \times [1..m] \times \{\uparrow, \downarrow\} \quad (59)$$

are, respectively, the sets of relation locations and transition rules that were used to extend $S$-systems into $L$-systems. The set $C_L$ is extending inputs and the set $R_L$ is extending the outputs of $S$-systems. We define the set $L_{obs}$ of $L$-observations by

$$L_{obs} = S_{obs} \times \text{bool}^{R_L}$$

and the set of $L$-behaviours by

$$L_{beh} = (L_{obs})^{(C \cup C_L)^*}. \quad (60)$$

We can also define $L$-traces. Given an $\sigma$ in $(C \cup C_L)^*$ and $\phi$ in $L_{beh}$, the $L$-trace determined by $\phi$ and $\sigma$ is defined by

$$\langle \phi(\varepsilon), \phi(\sigma[0]), \phi(\sigma[1]), \ldots, \phi(\sigma) \rangle. \quad (61)$$

Intuitively, we can also describe the behaviour of an $S$-system by restricting the behaviour of an $L$-system. This procedure is similar to what we did in Section 2.5.2, where we restricted $O$-behaviours to turn them into $M$-behaviours. More precisely, consider the restriction function $|_{C^*} : L_{beh} \rightarrow S_{beh}$, defined for $\phi$ in $L_{beh}$ and $\sigma$ in $C^*$ by

$$\phi|_{C^*}(\sigma) = \psi, \quad (62)$$

such that, for locations $f$ in $F$, and $r$ in $R$,

$$\psi(\sigma)(f, r) = \phi(\sigma)(f, r) \quad (63)$$

The function $|_{C^*}$ can be naturally applied by the environment, making $L$-systems produce $S$-traces instead of $L$-traces. More precisely, if the environment only provides inputs from $C$ and not from $C_L$, and we ignore the locations in $R_L$, then the $L$-behaviour $\phi$ is projected to the $S$-behaviour $\phi|_{C^*}$. Thus, we have shown how to specify $S$-systems from $L$-systems, and vice versa.
4.1 Introduction

In Deliverable 3.1.1. [13], the level 1 Interaction Machine was defined as a (synchronous) dynamic automata network comprising a time varying structure, generalizing the usual definition of automata network. Dually, we model Interaction Machines as collections of interdependent behaviours put together to display a joint behaviour. The two interpretations of Interaction Machines are compatible: each behaviour in the collection represents an automaton in the dynamic network. We want to model Interaction Machines as coalgebras, because that would allow us to define a behaviour-based specification language for them, which is the main objective of this deliverable.

The behaviours of Mealy machines that were introduced in Section 2.2.2 are not suitable for Interaction Machines, as those behaviours describe independent machines; that is, machine that only require environmental input to make a transition and to produce an output. Therefore, we need to formalise the concept of dependent Mealy machines; which are machines that require the output of other machines in order to produce an output of their own. With dependent Mealy machines modelled coalgebraically, it is possible to describe Level 1 Interaction Machines from a coalgebraic perspective.

4.2 $D$-Behaviours

As defined in Section 2.2.2, an $M$-behaviour is a function that maps non-empty sequences of inputs to a single output. In order to define Interaction Machines as coalgebras, our first objective is to define dependent $M$-systems and their behaviours. A dependent $M$-system is a system whose outputs not only depend on the current input and the current state, but they also depend on the outputs of other neighbour (dependent) $M$-systems.

Let $A$ be the set of inputs, $B$ be the set of outputs, $V$ be a set of indexes, and $\mathcal{P}_{\text{fin}}(V)$ be the set of finite subsets of $V$. We define the functor $\mathbb{D}$, for set $X$, by

$$\mathbb{D}(X) = \mathcal{P}_{\text{fin}}(V) \times (B \times X)^{(A \times B)^V}. \quad (64)$$

The functor $\mathbb{D}$ defines the category of dependent $M$-systems $\text{Set}_D$. A $\mathbb{D}$-coalgebra $\alpha$ is a function $X \xrightarrow{\alpha} \mathbb{D}(X)$ that maps an element $x$ in $X$ to the pair $(N_x, \delta_x)$. The finite set $N_x$ is the set of neighbours of $\alpha$ at state $x$. The function $\delta_x: A \times B^V \to B \times X$ is the transition function that maps a pair $(a, \rho)$ in $A \times B^V$ to a pair $(b, x')$ in $B \times X$. If $\delta_x(a, \rho) = (b, x')$, then we say that state $x$ transits to $x'$ and outputs $b$ when it receives the input $a$ and the $v^{th}$ neighbour of $\alpha$ has output $\rho(v)$.

By using the formulas in [9, Lemma 6], we define the set $\mathbb{D}_{\text{beh}}$ of $\mathbb{D}$-behaviours by

$$\mathbb{D}_{\text{beh}} = \left(\mathcal{P}_{\text{fin}}(V) \times B^{(A \times B)^V}\right)^{(A \times B)^V}. \quad (65)$$

A $\mathbb{D}$-behaviour should be interpreted as follows. Let $X \xrightarrow{\alpha} \mathbb{D}(X)$ be a $\mathbb{D}$-system and let $x$ be a state of $X$ such that $x$ has the $\mathbb{D}$-behaviour $\phi$. The $\mathbb{D}$-behaviour $\phi$ gives information about the neighbours of $\alpha$ at state $x$, as well
The restriction function in the pair $\phi(\varepsilon) = (N_\varepsilon, \theta_\varepsilon)$, with $N_\varepsilon = N_\varepsilon$. We remark that $\varepsilon$ is not a state of $\phi$ (or not a $\mathbb{D}$-system), so the set $N_\varepsilon$ refers to the set of neighbours that $\phi$ associates to the input sequence $\sigma$. The initial $\mathbb{D}$-observation gives out the following information:

- the current neighbours of $\alpha$ are indexed by the elements of $N_\varepsilon$, and
- with $a$ in $A$ and $\rho$ in $B^V$, a transition on pair $(a, \rho)$ with $\delta_\varepsilon(a, \rho) = (\theta_\varepsilon(a, \rho), x')$ causes the output $\theta_\varepsilon(a, \rho)$ to be observed, and $x$ to make a transition to state $x'$.

By defining $\mathbb{D}$-behaviours we are moving one step closer to the coalgebraic definition of Interaction Machines. We now need to find a final $\mathbb{D}$-system, so that we can define a behaviour-based specification language for dependent Mealy machines.

4.2.1 A Final $\mathbb{D}$-system

From Section 2.2.2 we know that the carrier sets of final systems can be used to define behaviour-based specification languages. Again, by using the formulas in [9, Lemma 6], we define the final $\mathbb{D}$-system $\mathbb{D}_{beh} \xrightarrow{\pi_D} \mathbb{D}(\mathbb{D}_{beh})$, for $\phi$ in $\mathbb{D}_{beh}$ with $\phi(\varepsilon) = (N_\varepsilon, \theta_\varepsilon)$, by

$$\pi_D(\phi) = (N_\varepsilon, \delta_\phi);$$

(66)

where $\delta_\phi$ in $(B \times \mathbb{D}_{beh})^{(A \times B^V)}$ is defined, for $(a, \rho)$ in $A \times B^V$, by

$$\delta_\phi(a, \rho) = (\theta_\varepsilon(a, \rho), \phi');$$

(67)

such that $\phi'$ in $\mathbb{D}_{beh}$ is defined, for $\sigma$ in $(A \times B^V)^*$, by

$$\phi'(\sigma) = \phi((a, \rho) \cdot \sigma).$$

(68)

The finality of $\pi_D$ allows us to claim that $\mathcal{P}(\mathbb{D}_{beh})$, the power set of $\mathbb{D}_{beh}$, is the set of dependent Mealy machine specifications or $\mathbb{D}$-specifications. These $\mathbb{D}$-specifications can be used to specify the components (or sub machines) of Interaction Machines.

There is a problem though: $\mathbb{D}$-behaviours may model systems that do not react to the outputs provided by their corresponding neighbours. Therefore, not all $\mathbb{D}$-behaviours model “correct” dependent Mealy machines. More precisely, a $\mathbb{D}$-behaviour may model a Mealy machine that ignores the outputs of some of its neighbours, or observes the outputs of machines that are not considered its neighbours. We do not want that. Thus, we need to perform some conditioning on $\mathbb{D}$-systems such that they behave in the way we expect them to. This process results in the creation of $\mathbb{D}^*$-systems, the subcategory of $\mathbb{D}$-systems that “behave correctly”.

4.2.2 $\mathbb{D}^*$-systems

Similar to the conditioning of $\text{ASML}$-systems that forces them to behave as ASMs (see Equation 31), we impose behavioural conditions on $\mathbb{D}$-systems so that they behave like the dependent Mealy machines that were described in Deliverable 3.1.1 [13].

With $X \xrightarrow{\alpha} \mathbb{D}(X)$ and $x$ in $X$, let $\alpha(x) = (N_x, \delta_x)$. We define the restriction function $\cdot|_{N_x} : B^V \rightarrow B^{N_x}$, for $\rho$ in $B^V$ and $v$ in $N_x$, by

$$\rho|_{N_x}(v) = \rho(v).$$

(69)

The restriction function $\cdot|_{N_x}$ simply changes the signature of functions from $V \rightarrow B$ to $N_x \rightarrow B$. Now, let $\rho_1$ and $\rho_2$ in $B^V$. If $\rho_1|_{N_x}$ is equal to $\rho_2|_{N_x}$, then we say that the output sequences $\rho_1$ and $\rho_2$ are $x$-equivalent.
In other words, the outputs of the neighbours of $\alpha$ at $x$ are the same in $\rho_1$ and in $\rho_2$. The outputs of machines that are not neighbours of $\alpha$ at state $x$ can be anything. In order to force $\mathbb{D}$-systems produce observations based on their neighbours, we condition the transition function $\delta_x$ to be equal for every $x$-equivalent element in $B^V$. Formally,
\[
\forall a \in A, \forall \rho_1, \rho_2 \in B^V. (\rho_1|_{N_x} = \rho_2|_{N_x} \Rightarrow \delta_x(a, \rho_1) = \delta_x(a, \rho_2))
\] (70)
We define the subcategory $\text{Set}_{\mathbb{D}^*}$ with the $\mathbb{D}$-systems that satisfy Equation 70. We refer to those systems as $\mathbb{D}^*$-systems, and they represent the dependent Mealy machines that behave correctly. Equation 70 can be considered to restrict $\mathbb{D}$-systems in a similar way that the behavioural assertion $x.ch(n).bal = x.bal + n$ is restricting the objects of the class $\text{BankAccount}$, shown in Section 1.2.3.

Consider a $\mathbb{D}^*$-system $X \xrightarrow{\alpha} \mathbb{D}^*(X)$ and a state $x$ in $X$ such that $\alpha(x) = (N_x, \delta_x)$. We define the function $\delta^*_x: A \times B^{N_x} \to B \times X$, for $a$ in $A$ and $\rho$ in $B^V$, by
\[
\delta^*_x(a, \rho|_{N_x}) = \delta_x(a, \rho).
\] (71)
The transition function $\delta^*_x$ removes the redundancy in $\delta_x$, and it is useful for implementation purposes. This can be done because the transition function of $\mathbb{D}^*$-systems depends only on the outputs of the machines in the current neighbourhood. Given that the neighbourhood is always finite, a process computing the transition function eventually terminates.

### 4.2.3 $\mathbb{D}^*$-behaviours

If $\mathbb{D}^*$-systems are a subset of $\mathbb{D}$-systems, then we can naturally assume that only a subset of $\mathbb{D}$-behaviour is realised by them. Let $\phi$ be an element of $\mathbb{D}_{beh}$. We say that $\phi$ is a $\mathbb{D}^*$-behaviour if, for all $\sigma$ in $(A \times B^V)^*$ with $\phi(\sigma) = (N_\sigma, \theta_\sigma)$, the condition
\[
\forall a \in A, \forall \rho_1, \rho_2 \in B^V. (\rho_1|_{N_\sigma} = \rho_2|_{N_\sigma} \Rightarrow \theta_\sigma(a, \rho_1) = \theta_\sigma(a, \rho_2))
\] (72)
is satisfied. In other words, if $\phi$ is a $\mathbb{D}^*$-behaviour, then the output function $\theta_\sigma$ can be calculated by using the input $a$ and the outputs of the neighbours indexed by $N_{\sigma}$, ignoring the outputs of machines that are not neighbours. We call the set $\mathbb{D}^*_{beh}$ the set of behaviours of correctly behaved dependent Mealy machines, and we define the set $\mathbb{P}(\mathbb{D}^*_{beh})$ to be the set of $\mathbb{D}^*$-specifications, which contains all behaviour-based specifications for correctly behaved dependent Mealy machines.

#### Example 1 (The $n^{th}$ Component of a Binary Number)
Let the indexing set $V$ be equal to $\mathbb{N}$: (the set of natural numbers), $A = \{0, 1\}$ be the set of inputs, and $B = \{0, 1\}$ be the set of outputs. Given $n$ in $V$, we define the $\mathbb{D}^*$-behaviours $\phi^0_n$ and $\phi^1_n$, for $\sigma$ in $(A \times B^V)^*$, by
\[
\phi^0_n(\sigma) = (N^0_\sigma, \theta^0_\sigma),
\] (73)
\[
\phi^1_n(\sigma) = (N^1_\sigma, \theta^1_\sigma),
\] (74)
with $N^0_\sigma = \{0, \ldots, n-1\} = N^1_\sigma$, and $\theta^0_\sigma$ and $\theta^1_\sigma$ defined, for $(a, \rho)$ in $A \times B^{\{0,\ldots,n-1\}}$, by
\[
\theta^0_\sigma(a, \rho) = \begin{cases} 
0, & \text{if } a = 0; \\
0, & \text{if } a = 1 \text{ and } \exists i \in N^0_\sigma \rho(i) = 0; \\
1, & \text{if } a = 1 \text{ and } \forall i \in N^0_\sigma \rho(i) = 1.
\end{cases}
\] (75)
\[
\theta^1_\sigma(a, \rho) = \begin{cases} 
1, & \text{if } a = 0; \\
1, & \text{if } a = 1 \text{ and } \exists i \in N^1_\sigma \rho(i) = 0; \\
0, & \text{if } a = 1 \text{ and } \forall i \in N^1_\sigma \rho(i) = 1.
\end{cases}
\] (76)
Note that $\phi^0_n$ and $\phi^1_n$ are constant for $\sigma$. The function $\phi^0_n$ describes the behaviour of the $n^{th}$ component of a binary number with value 0. The inputs correspond to adding 0 or adding 1 to a binary number. In this sense, the $n^{th}$ component of a number changes from 0 to 1 only if a 1 is received and all the previous components are 1. Similarly, the function $\phi^1_n$ describes the behaviour of the $n^{th}$ component of a binary number, but this time with value 1.
4.3 Level 1 Interaction Machines, Coalgebraically

Now that we have formally defined a way to represent the behaviour of dependent Mealy machines, we can define the components of Interaction Machines. We define the functor $\mathbb{IM}$, for set $X$, by

$$\mathbb{IM}(X) = (1 + \mathbb{D}_{beh}^*)^V \times \left( (1 + B)^V \times X \right)^A,$$

(77)

where $1$ is the set $\{\perp\}$ and $+$ is disjoint union of sets. The functor $\mathbb{IM}$ defines the category of $\mathbb{IM}$-systems or Level 1 Interaction Machines. Let $x$ in $X$ and $X \xrightarrow{\alpha} \mathbb{IM}(X)$ be an $\mathbb{IM}$-system. The $\mathbb{IM}$-system $\alpha$ “unpacks” $x$ into a pair

$$\alpha(x) = (\Psi_x, \Delta_x);$$

(78)

where $\Psi_x$ is the topology at state $x$ and $\Delta_x$ is the global transition function at state $x$.

The topology is interpreted as follows: given $v$ in $V$, the value $\Psi_x(v)$ can be either $\perp$ or a $D^*$-behaviour. If $\Psi_x(v)$ is $\perp$, then the index $v$ is free; that is, no system is indexed by $v$. Conversely, if $\Psi_x(v)$ is a $D^*$-behaviour, then that behaviour has index $v$ and belongs to the network of the Interaction Machine $\alpha$. Thus, we consider $\Psi_x$ to be a partial function from $V$ to $\mathbb{D}_{beh}^*$, and its domain $\text{dom}(\Psi_x)$ is the set of indexes of the dependent machines present in the network at state $x$.

The transition function is interpreted as follows: given $a$ in $A$, if $\Delta_x(a) = (\rho, x')$, then we say that the state $x$ transits to the state $x'$ on input $a$, and it produces a sequence of outputs $\rho$.

Having a functor brings us one step closer to a way of representing the behaviours of Interaction Machines. We remind the reader that there are three problems surrounding the definition of a behaviour-based specification language: representing behaviours, combining behaviours and comparing behaviours. In the following sections, we explain how we solved them for the specification of Interaction Machines.

4.3.1 $\mathbb{IM}$-Behaviours

Again, by using the formulas in [9, Lemma 6], we define the set $\mathbb{IM}_{beh}$ of $\mathbb{IM}$-behaviours by

$$\mathbb{IM}_{beh} = \left( (1 + \mathbb{D}_{beh}^*)^V \times \left( (1 + B)^V \right)^A \right)^{A^*}.$$

An $\mathbb{IM}$-behaviour $\Phi$ is a function that, given $\sigma$ in $A^*$, it maps the input sequence $\sigma$ to a pair $(\Psi_\sigma, \Theta_\sigma)$ in $(1 + \mathbb{D}_{beh}^*)^V \times \left( (1 + B)^V \right)^A$. If $\Phi(\sigma) = (\Psi_\sigma, \Theta_\sigma)$, then we say that $\Psi_\sigma$ is the topology after $\sigma$ and $\Theta_\sigma$ is the global output function after $\sigma$.

Once more, by using the formulas in [9, Lemma 6], we define the final $\mathbb{IM}$-system $\mathbb{IM}_{beh} \xrightarrow{\pi_{IM}} \mathbb{IM}(\mathbb{IM}_{beh})$, for $\Phi$ in $\mathbb{IM}_{beh}$ with $\Phi(\varepsilon) = (\Psi_\varepsilon, \Theta_\varepsilon)$, by

$$\pi_{\mathbb{IM}}(\Phi) = (\Psi_\Phi, \Delta_\Phi);$$

(79)

with $\Psi_\Phi = \Phi_\varepsilon$ and $\Delta_\Phi$ defined, for $a$ in $A$, by

$$\Delta_\Phi(a) = (\Theta_\varepsilon(a), \Phi');$$

(80)

and $\Phi'$ in $\mathbb{IM}_{beh}$ defined, for $\sigma$ in $A^*$, by

$$\Phi'(\sigma) = \Phi((a) \cdot \sigma).$$

(81)

By finality of $\pi_{\mathbb{IM}}$, we can consider the elements of $\mathbb{IM}_{beh}$ to represent the behaviours of $\mathbb{IM}$-systems, including those of Interaction Machines. However, the set $\mathbb{IM}_{beh}$ also contains the behaviours of Interaction Machines that are not semantically correct, in a similar way that the set $\mathbb{D}_{beh}$ contained the behaviours of dependent Mealy machines that did not behave correctly. In this sense, we need to impose behavioural conditions of $\mathbb{IM}$-systems such that they behave like correct Interaction Machines. Those behavioural conditions are what we call the interaction conditions.
4.3.2 The Interaction Conditions

Let \( a \) in \( A \). If the input \( a \) is received by an \( IM \)-system, then two things can happen:

- the topology can change. More precisely, for each index \( v \) in \( V \), a \( \mathbb{D}^* \)-behaviour may appear, disappear or be replaced by another.
- an output sequence (i.e., the global output) is produced.

As we previously stated, not all \( IM \)-systems behave like Interaction Machines. Thus, we define some behavioural conditions to tell them apart. Let \( x \) in \( X \) and \( X \overset{\alpha}{\rightarrow} IM(X) \) be some \( IM \)-system with \( \alpha(x) = (\Psi_x, \Delta_x) \). We say that \( \alpha \) is an Interaction Machine if it satisfies the Interaction Conditions. Those conditions are:

1. (Finiteness.) The topology is finite, at all times;
2. (Connectivity.) Dependent components are connected to the components they depend on, even if those components change, at all times;
3. (Consistency.) The output of the Interaction Machine matches the outputs of the components, at all times.

Formally:

1. (Finiteness.) The set \( V_x \) of indexed systems at state \( x \), defined by \( V_x = \{ v \in V \mid \Psi_x(v) \neq \bot \} \), is finite;
2. (Connectivity.) If \( \phi \) is the \( v^th \) \( \mathbb{D}^* \)-behaviour with \( \phi(\varepsilon) = (N_{\varepsilon}, b_{\varepsilon}) \), then, for all \( \psi' \) in \( N_{\varepsilon} \), the \( \Psi_{\psi'}(\psi') \) is an element of \( V_x \). In other words, the neighbours of \( \phi \) are in the topology, and
3. (Consistency.) For all \( a \) in \( A \), if \( \Delta_x(a) = (\rho, x') \) and \( b \) is the output of the \( v^th \) \( \mathbb{D}^* \)-system on input \( a \), then \( \rho(v) \) is equal to \( b \). Additionally, if \( \Psi_{\psi'}(\psi') = \bot \), then \( \rho(v) = \bot \). In other words, the global output of the IM is determined by the output of the internal components in the topology.

By using the Interaction Conditions, we can define the subcategory of \( IM^* \)-system, containing the \( IM \)-systems that satisfy the Interaction Conditions. Note that this process is very similar to what was done for \( D \)-systems, but this time we impose three behavioural conditions, not just one.

We also adapt the Interaction Conditions such that we can determine if an \( IM \)-behaviour represents the behaviour of an Interaction Machine. Let \( \Phi \) in \( IM_{beh}^* \). We say that the \( IM \)-behaviour is an \( IM^* \)-behaviour or an Interaction Machine behaviour if, for all \( \sigma \) in \( A^* \) with \( \Phi(\sigma) = (\Psi_\sigma, \Theta_\sigma) \), the following conditions are satisfied:

1. (Finiteness.) The set \( V_\sigma \), defined by \( V_\sigma = \{ v \in V \mid \Psi_x(v) \neq \bot \} \), is finite;
2. (Connectivity.) For all \( v \) in \( V_\sigma \), if \( \Psi_\sigma(v)(\varepsilon) = (N_{\varepsilon}, \theta_{\varepsilon}) \), then \( N_{\varepsilon} \) is a subset of \( V_\sigma \);
3. (Consistency.) If \( \sigma = \sigma' \cdot (\langle a \rangle) \), then, for all \( a \) in \( A \) and all \( v \) in \( V_\sigma \),
   \[
   \theta_v(a, \Theta_\sigma(\langle a \rangle)_{|N_v}) = \Theta_\sigma(a)(v);
   \] (82)
   also, if \( \Psi_\sigma(v') \) is equal to \( \bot \) for some \( v' \) in \( V \), then, for all \( a \) in \( A \), we require that \( \Theta_\sigma(a)(v') = \bot \).

Let \( IM_{beh}^* \) be the set of \( IM_{beh}^* \). With the definition of the set \( IM_{beh}^* \) we have solved the representation problem of Interaction Machine behaviours. Given that \( IM_{beh}^* \) is a subset of the set of functions \( \left( (1 + \mathbb{D}_{beh}^*)^V \times (1 + B)^N \right)^A \), comparing Interaction Machine behaviours can be reduced to comparing functions in such set. Finally, to solve the problem of combining Interaction Machine behaviours, we define the set \( P(IM_{beh}^*) \) to be the behaviour-based specification language for Interaction Machines, so the combination of two \( IM^* \)-behaviours \( \Psi_1 \) and \( \Psi_2 \) yields the Interaction Machine specification \( \{ \Psi_1, \Psi_2 \} \). Note that the specification language \( P(IM_{beh}^*) \) is functional. With the following example, we illustrate how to specify a single Interaction Machine by using the behaviour-based specification language.
Example 2 (An Interaction Machine for Binary Numbers). Consider the $\mathbb{D}^*$-behaviours defined in Example 1. With $V = \text{nat}$, $A = \{0, 1\}$, and $B = \{0, 1\}$, we inductively define the $\mathbb{M}^*$-behaviour $\Phi_{\text{bin}}$, for $\sigma$ in $A^*$, by

$$
\Phi_{\text{bin}}(\sigma) = (\Psi_\sigma, \Theta_\sigma),
$$

(83)

with $\Psi_\sigma$ defined, for $v$ in $V$, by

$$
\Psi_\sigma(v) = \begin{cases} 
\phi_0^0, & \text{if } \sigma = \varepsilon \text{ and } v = 0; \\
\bot, & \text{if } \sigma = \varepsilon \text{ and } v \neq 0; \\
\Psi_{\sigma'}(v), & \text{if } \sigma = \sigma' \cdot \langle 0 \rangle, \text{ with } \sigma' \in A^*; \\
\Psi_{\sigma'}(v), & \text{if } \sigma = \sigma' \cdot \langle 1 \rangle \text{ and } \exists i < v. \Psi_{\sigma'}(i) \neq \phi_1^i; \\
\phi_0^1, & \text{if } \sigma = \sigma' \cdot \langle 1 \rangle \text{ and } \forall i < v. \Psi_{\sigma'}(i) = \phi_0^1 \text{ and } \Psi_{\sigma'}(v) \neq \phi_1^v; \\
\phi_1^0, & \text{if } \sigma = \sigma' \cdot \langle 1 \rangle \text{ and } \forall i < v. \Psi_{\sigma'}(i) = \phi_1^1 \text{ and } \Psi_{\sigma'}(v) = \phi_0^v.
\end{cases}
$$

(84)

We define $\Theta_\sigma$ by using the consistency condition: assuming $\sigma = \sigma' \cdot \langle a' \rangle$, for $a$ in $A$ and $v$ in $V$ with $\Psi_\sigma(v)(\varepsilon) = (N_v, \theta_v)$,

$$
\Theta_\sigma(a)(v) = \begin{cases} 
\theta_v(a, \Theta_{\sigma'}(a')|_{N_v}), & \text{if } \Psi_\sigma(v) \neq \bot; \\
\bot, & \text{otherwise}.
\end{cases}
$$

(85)

Given this definition, an $\mathbb{M}^*$-system realising $\Phi$ starts by only indexing the $\mathbb{D}^*$-behaviour $\phi_0^0$. Whenever a 0 is received, each indexed behaviour remains the same. However, whenever a 1 is received, the smallest index $v$ that is not indexing a behaviour $\phi_0^v$ gets changed to $\phi_1^v$, and every behaviour indexed by $i$ such that $i$ is smaller than $v$ changes to $\phi_0^i$.

Note that the number of machines indexed by is always finite, so $\Phi$ satisfies the finiteness condition. Additionally, the indexes of the neighbours of each indexed behaviour are also indexing a behaviour; satisfying the connectivity condition. Thus, $\Phi_{\text{bin}}$ is in effect an $\mathbb{M}^*$-behaviour, and specifies behaviour-wise the Interaction Machine that describes what happens when a 0 or a 1 is added to a binary number.
Chapter 5

Conclusion and Future Work

In Chapter 2 we presented a brief introduction to behaviour-based specification languages. If $A$ is a set of inputs and $B$ is a set of outputs, then, behaviour-wise, Mealy machines and Moore machines are specified by the elements of the sets $B^{A^+}$ and $B^{A^*}$, respectively. A behaviour-based specification for a multi-purpose Mealy machine is then a subset of $B^{A^+}$, where $B^{A^+}$ is the behaviour-based specification of a universal Mealy machine. This paves the way towards a specification language of Interaction Machines that uses functions to describe their functionalities.

In Chapter 3 we showed that basic ASMs can be modelled coalgebraically by proposing the polynomial functor $ASM$ described in Equation 29. The functor $ASM$ defines the category of $ASM$-systems, and allows the precise definition of the notions of $ASM$-bisimulations, $ASM$-observations, $ASM$-behaviours, $ASM$-traces, $ASM$-programs, and $ASM$-implementations. We also showed that ASMs can be seen as conditioned Moore systems whose outputs are Tarski structures and whose inputs are semantically charged.

In Chapter 4, we modelled Interaction Machines as functions of the set $\left[(1 + D_{beh}^+)V \times \left((1 + B)V\right)^A\right]^{A^*}$, where $D_{beh}^+$ is the set of correct, dependent Mealy machine behaviours, $1$ is the set $\{\bot\}$, $+$ is disjoint union of sets and $V$ is a set of (potentially infinite) indexes. However, not all functions of that set are Interaction Machines; only the functions that satisfy the Interaction Conditions can be considered to be behavioural specifications of Interaction Machines. The Interaction Conditions are:

1. (Finiteness.) The topology is finite, at all times;
2. (Connectivity.) Dependent components are connected to the components they depend on, even if those components change, at all times;
3. (Consistency.) The output of the Interaction Machine matches the outputs of the components, at all times.

We also provide an example of an Interaction Machine specification that models the binary representation of natural numbers.

In conclusion, being able to describe the behaviours of Interaction Machines as functions is a great step towards the definition of a more friendly specification language for Interaction Machines, because these functions define templates that can be filled by the users that want to specify Interaction Machines. We will now focus on making the definition of an Interaction Machine an interactive process between the specification framework and the user. As the user fills the templates, the specification framework will enforce the Interaction Conditions, so if the user makes a mistake, the framework will report the errors and let the user know what went wrong. For example, if the user forgets to connect a component to one of its dependencies, the Interaction Condition of connectivity is not satisfied, and the framework will report it. Once the framework endorses the specification given by the user, the resulting system can be considered an Interaction Machine. More insights will be provided in Deliverables 5.1 and 4.2.
References


