Constructing Runtime Models with Bigraphs to Address Ubiquitous Computing Service Composition Volatility

Submitted for the Degree of Doctor of Philosophy, University of Sussex, November 2014

Renan Krishna

Dedicated to the memory of my father late Professor M.M. Krishna

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

First of all, I would like to express my sincere gratitude to my supervisor Dr. Ian Wakeman for his invaluable guidance, help and support during the course of my work.

Thanks are also due to Dr. Dan Chalmers for his comments and suggestions during the course of my work.

A big thank you to Dr. Des Watson, Dr. Bernhard Reus, Dr. Martin Berger, Dr. Ian Mackie, and Dr. George Parisis for providing invaluable suggestions and encouragement whenever they were needed.

Thanks are also due to the late Robin Milner for the enthusiasm he showed for my work.

I would like to express my appreciation to both the present and past members of Foundations of Software Systems Group namely: Dr. Jon Robinson, Dr. Jian Li, Dr. Roya Feizy, Dr. Lachhman Dhomeja, Dr. Yasir Malkani, Dr. Anirban Basu, Dr. Stephen Naicken, Dr. James Stanier, Dr. Simon Fleming, Dr. Aesha Alsiyami, Dr. Danny Matthews, Dr. Ben Horsfall, Thomas Harvey, Ciaran Fisher and Rakan Abdullah Alsowail for being a lively company both in the lab and outside.

Many thanks to Christopher Tucker and Alex Castellanos for the fantastic time I had working for the University of Sussex Residential Services and living in East Slope during the course of my work.

Thanks also to all my friends for being the lovely and supportive people they are.

Lastly but most importantly, a big thank you to my family: to my mother Dr. Rama Nigam who has solidly believed in me and to my sister Dr. Delfi Krishna who has been an inspiration.

I would like to dedicate this thesis to the memory of my father late Professor M.M. Krishna who I am sure would have been proud of my work.

ABSTRACT

In this thesis, we explore the appropriateness of the language abstractions provided by Bigraphs to construct a model at runtime to tackle the problem of volatility in a service composition running on a mobile device.

Our contributions to knowledge are as follows:

1) We have shown that Bigraphs (Milner, 2009) are suitable for expressing models at runtime.
2) We have offered Bigraph language abstractions as an appropriate solution to some of the research problems posed by the models at runtime community (Aßmann et al., 2012).
3) We have discussed the general lessons learnt from using Bigraphs for a practical application such as a model at runtime.
4) We have discussed the general lessons learnt from our experiences of designing models at runtime.
5) We have implemented the model at runtime using the BPL Tool (ITU, 2011) and have experimentally studied the response times of our Bigraphical model. We have suggested appropriate enhancements for the tool based on our experiences.

We present techniques to parameterize the reaction rules so that the matching algorithm of the BPL Tool returns a single match giving us the ability to dynamically program the model at runtime. We also show how to query the Bigraph structure.

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

1.1 THE CONTEXT OF OUR STUDY: A SCENARIO

Ubiquitous computing systems (Weiser, 1999) are often characterized as being volatile (Coulouris, 2012). This includes all of the following properties: 1) Device and communication link failures, 2) Variation in the properties of communication such as bandwidth, and 3) Creation and destruction of associations which are logical communication relationships between software components resident on the devices.

Imagine a scenario in which a user, Alice, is strolling around in a shopping mall with a mobile device running a composition of services being offered by devices that are embedded all over the mall. She might want to buy a pair of jeans but wants to compare prices, find the location of nearby shops and check customer ratings of the shops. The (volatile) composite service running on her mobile device will be comprised of a price comparison service, a location service and a service offering customer ratings of the shops in the mall. We assume that Alice's device has various forms of wireless connectivity (Bluetooth, Wi-Fi, 3G etc.). As Alice moves around in the shopping mall, the mobile device running the composition will suffer disconnections to some of the services due to radio occlusions, multi-hop wireless routing, or the user moving 'out of range' (Coulouris, 2012). These same factors could also lead to a highly varying latency and bandwidth of the connection between a service and a mobile device. Moreover, the user's device might run out of battery. If one of the services disappears (malfunctions) because of this volatility, we want our system to replace it with an equivalent service(s) without any user intervention.

The volatility in service composition arises from changes in context i.e. changes in the external environment. However, the effect of volatility is on the internal working of the service composition. Volatility may result in a higher level of complexity as services participating in the service composition may appear and disappear at a high rate and break their interconnection with the service composition. There might also be a large number of 'equivalent' services to choose from.

Leveraging software models to inform runtime adaptation mechanisms has become an important technique to manage the complexity of evolving software as it executes (Aßmann et al., 2012, Blair et al., 2009, France and Rumpe, 2007). This is because models at runtime provide "*abstractions of runtime phenomenon"* (France and Rumpe, 2007) rather than abstractions of

design time artifacts. We wish to model at runtime a service composition running on a mobile device.

Software models at runtime can be expressed with Bigraphs (Milner, 2009). Bigraphs are graphical structures with nodes and edges. These nodes can be placed inside each other and be linked with edges. Bigraphs have been rigorously formalized with category theory (Barr and Wells, 1990). They have been shown to capture the theory of Petri nets (Milner, 2004a), pi-calculus (Milner, 1999), CCS (Milner, 2006a), mobile ambient (Jensen, 2006) and lambda calculus (Milner, 2004b). Because of their graphical structure, Bigraphs can be intuitively used to visualize and model physical and virtual structures having location, communication and behavior (Greenhalgh, 2009b). This thesis, to the best of our knowledge, is the first work to explore the appropriateness of the language abstractions provided by Bigraphs to construct a model at runtime to tackle the problem of volatility in a service composition running on a mobile device.

1.2. THE RESEARCH QUESTION

Our thesis answers the following research question:

Are the language abstractions provided by Bigraphs sufficient and appropriate to construct a model at runtime to tackle the problem of volatility in a service composition running on a mobile device?

This question combines two major issues that have not been addressed in the literature:

i. *How do we use Bigraphs to construct a model at runtime?*
ii. *Do Bigraphs offer the appropriate language abstractions to address the open research questions being explored by the models at runtime community?*

The two caveats on the scope of the above question are that firstly, we will not replicate all programming language abstractions with our Bigraphical model at runtime. Instead, we will abstract upon only some selected elements of the service composition. Secondly, we would be accessing the control constructs of SML through MiniML (a subset of Standard ML-see Chapter 2) since Bigraphs lack control structures.

1.3. APPLICATIONS OF BIGRAPHS AND MODEL AT RUNTIME

We now discuss the reasons as to why it is worthwhile to answer the question discussed in the previous section.

Firstly, Bigraphs have been envisaged as a step towards tackling the complexity of ubiquitous systems (Milner, 2009) . If found useful, Bigraphs models could be used as foundational models in a 'tower of models'(Milner, 2008b). In such a tower, the higher-level models will *express* concepts such as trust and the lower level models will *implement* concepts such as trust by for example having an agent accept data only from a 'trustworthy' agent.

Secondly, the models-at-runtime research community envisages using Model-Driven-Engineering techniques to develop models that are abstractions of runtime phenomenon (Blair et al., 2009). If found useful, such models could be used to support reasoning, dynamic state monitoring and control of systems at runtime.

1.4. OVERVIEW OF THE MAIN CONTRIBUTIONS

We now give a brief overview of the main contributions of this thesis.

1) This thesis responds to a call by Robin Milner (Milner, 2009) to explore the appropriateness of using Bigraphs in practical applications. We have successfully constructed a model at runtime with Bigraphs.

2) This thesis also responds to some of the research questions posed by the models at runtime community at the Dagstuhl seminar (Aßmann et al., 2012) by offering Bigraphical language abstractions as an appropriate solution.

3) We discuss the general lessons learnt from using Bigraphs for a practical application such as a model at runtime.

4) We discuss the general lessons learnt from our experiences of designing models at runtime.

5) We have implemented the model at runtime using the BPL Tool (ITU, 2011) and suggest appropriate enhancements for the tool.

1.5 THESIS STRUCTURE

We have organized the structure of this thesis along the following lines:

We start off in Chapter 2 where we give the background of our thesis. We discuss the literature related to our implementation's language- Bigraphs; our implementation's architecture- models at runtime; the example property- volatility; and finally the example system: - a ubiquitous computing service composition.

Having set the stage thus, in Chapter 3 we first describe our research question. Then we go on to discuss the requirements for our design and the design space for tackling the problem of volatile service composition.

Next, in Chapter 4, we discuss the architecture that we have used for our system and show how we have used the Bigraphical structure and reaction rule abstractions to program our system using the MiniML language (a subset of Standard ML) supported by the BPL Tool (ITU, 2011).

Then, in Chapter 5, we discuss our implementation approach, the functions that modify/access the two layers of our model and finally the functions that encapsulate adaptation logic and simulate test runs.

In Chapter 6, we present both a qualitative and quantitative evaluation of our implementation. The quantitative evaluation is done by loading our system that runs on a laptop with workload events. These workload events are generated on an Android machine running simulations based on the Shopping Mall Mobility model (Galati et al., 2013) and are sent to the laptop via a TCP connection. This quantitative evaluation focuses on testing if our Bigraphical model at runtime can be in-sync with the real world.

We conclude our thesis with Chapter 7. In that chapter, we discuss how this thesis answers the research question posed in Chapter 3; and describe our contributions to knowledge and future work.

2 BACKGROUND

2.1 INTRODUCTION

This chapter presents a thorough review of the background material relevant to our thesis. As discussed in the previous chapter, our thesis explores the appropriateness of Bigraph's abstractions to construct a model at runtime to tackle the problem of volatile service composition running on a mobile device. Therefore, we discuss the literature representing work being done by the research communities in Bigraphs, models at runtime, as well as the description of the property of volatility and faults occurring in a service composition running on ubiquitous computing systems. We have organized this chapter as follows:

In section 2.2, we start with a brief introduction to Bigraphs (Milner, 2009). We also discuss Plato-Graphical models (Birkedal et al., 2006) which is a minor extension of Bigraphs and has been used by us to inform our design of a two-layered model. Next, we present a brief introduction to the BPL Tool (ITU, 2011) that we have used to implement our system. Then, we discuss the literature presenting practical implementation of Bigraphs and the open research questions that stem from it. Finally, we discuss those ideas on the use of Bigraphs from the literature that constitute the take-off point for our thesis.

In section 2.3, we discuss the research work being done by the models at runtime community. Firstly, we give an explanatory background of models at runtime. Then, we discuss the literature presenting the use of architectural models at runtime and the open research questions that stem from it. Finally, we discuss those ideas on the use of architectural models at runtime from the literature that constitute the take-off point for our thesis.

Since volatility is the example property that our system deals with, in section 2.4, we discuss the relevant literature that characterizes volatility in ubiquitous systems.

Lastly, as our example system issue is faults occurring in a service composition running on a mobile device, in section 2.5, we discuss the taxonomy of faults that can occur in such systems.

2.2 BIGRAPHS: A NEW LANGUAGE

Bigraphs unify various process algebra and represent a Ubiquitous Abstract Machine that *"much like Von-Neumann's register machines could be utilized to build a tower of models for the complex concepts involved in ubiquitous computing"* (Milner, 2006b), (Birkedal et al., 2006). The notions of locality, mobility, connectivity and stochastics are captured in the theory of Bigraphs. A Bigraph model *"can be presented graphically for less technical clients and*

mathematically for analysts" (Milner, 2008a). The model also *"underlies a design methodology for engineers and provides an executable subset that is a programming language"* (Milner, 2008a).

In the following sub-sections, we firstly give an explanatory background for Bigraphs. Secondly, we describe what is being modelled with Bigraphs. Thirdly, we discuss the open research questions on the practical implementation of Bigraphs and finally we discuss the take-off point of our thesis with respect to Bigraphs.

2.2.1 EXPLANATORY BACKGROUND FOR BIGRAPHS

We now start with an informal discussion of the mathematics of Bigraph theory. This discussion is meant for someone who is interested in the practical usage of Bigraphs. Next, we also discuss the Bigraph Programming Language (BPL) Tool. We then discuss some of the syntax of the SUGAR module of the BPL Tool that we have used in this thesis. Finally, we give a brief overview of using the SUGAR module to express the Plato-graphic model that we have used in this thesis.

2.2.1.1 MATHEMATICAL BACKGROUND FOR BIGRAPHS

We now present a brief overview of the mathematical concepts used to define Bigraphs (Milner, 2008a, Milner, 2009). We do not present the full mathematical theory that defines the properties of Bigraphs. For a through treatment, the reader is referred to Milner's book (Milner, 2009). What follows is based on that book's first chapter and Milner's paper (Milner, 2008a) which is meant for practitioners who are interested in using Bigraphs for implementing real-world applications.

A general definition of Category (Milner, 2009): A Category C has a set of *objects* and a set of *arrows*. Milner denotes objects by *I*, *J*, *K* and arrows by *f*, *g*, *h* and we will follow his convention in this discussion. Each arrow *f* has a *domain* and *codomain* , which are both objects; if these are *I* and *J* then the notation used is $f : I \to J$, where *I* = dom(*f*) and J = cod(*f*). The set of arrows $f : I \to J$ is called the *homset* of *I* and *J* , and is written as $\mathbf{C}(I \to J)$ or simply $(I \to J)$.

Notations and Conventions (Milner, 2009):

Disjoint Sets: Let the set *A* and *A'* be two disjoint sets. Then the union of these two sets will be denoted by:

$$A \uplus A'$$

Finite Ordinal: A non-negative integer k will be considered a finite ordinal:

$$k = \{0,1,\ldots k\text{-}1\}$$

A category whose objects and arrows are finite ordinals and the maps between them will be represented by ORD.

Graph: A graph consists of nodes V and edges E. An edge joins a pair of nodes.

Hypergraph: is a generalization of a *graph* in which the edge may join any number of nodes.

Consider a *hypergraph* where each node $v \in V$ has an arity *ar(v)* which is a finite ordinal. Let the hyper graph have ports defined as:

$$P_V \stackrel{\text{def}}{\equiv} \biguplus_{v \in V} P_v.$$

Then a *hypergraph* is defined as a quadruple:

$$(V, ar, E, link)$$

where $ar : V \to ORD$ defines the arities, and link: $P_V \to E$ each port to an edge.

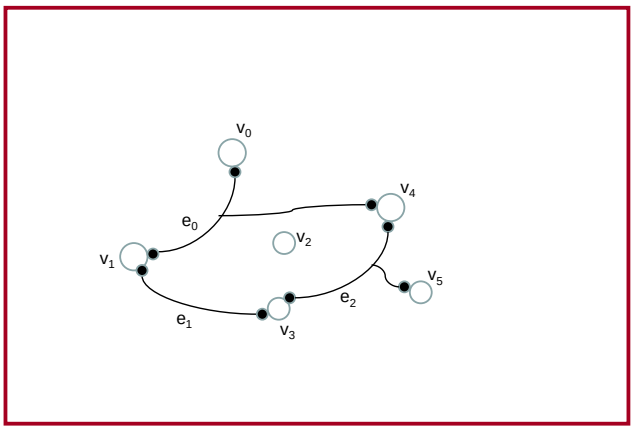

FIGURE 2-1: A HYPERGRAPH(MILNER, 2008A).

A hypergraph with nodes $\{v_0,\ldots..,v_5\}$ and edges E = { e_0, e_1, e_2} is shown in Figure 2-1. The figure represents nodes as circles, ports as black blobs, and edges as linkages between the ports.

In the Bigraph theory, these hypergraphs are enhanced into Bigraphs in four steps (Milner, 2008a).

a) The hypergraph is considered *linking* the nodes. The nodes are furnished with additional structure called *placing*. Because we have two structures in the graph, the prefix 'bi' is used with the word graph.

b) To make parts of the Bigraph externally accessible, *interfaces* are introduced.

c) To classify the nodes, *signatures* are then introduced.

d) Finally, operations to construct larger Bigraphs from smaller ones are defined.

2.2.1.1.1 BIGRAPHS IN PICTURES

Following Milner's paper (Milner, 2008a), we discuss each of the above four steps:

Placing and Linking: *"A Bigraph with nodes V and edges E has a hypergraph with nodes V and edges E, and a forest with nodes V"* (Milner, 2008a).

Nesting is allowed for the nodes. This spatial structure is called placing and is completely independent of the linking structure represented by the hypergraph. Thus, placing consists of a set of trees i.e. a forest of the nodes. The Bigraphs described so far are called *bare Bigraphs*. Milner uses the notation $\breve{F}$, $\breve{G}$, … to represent these bare Bigraphs (Milner, 2008a). Figure 2-2 shows a bare Bigraph $\breve{G}$ that has nodes $V = \{v_0,\ldots,v_5\}$ and edges $E = \{ e_0, e_1, e_2\}$, with its forest and hypergraph.

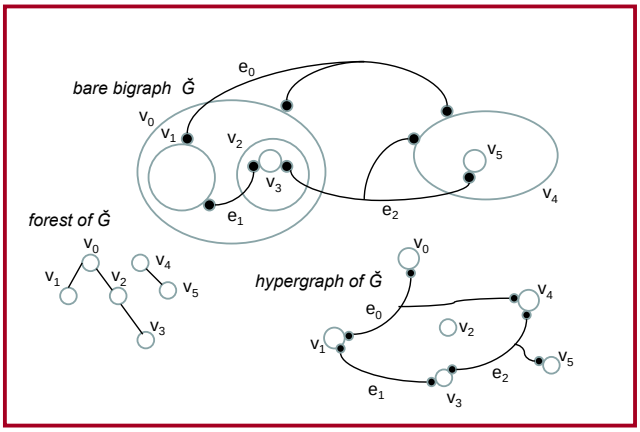

FIGURE 2-2: THE UPPER DIAGRAM PRESENTS BOTH THE FOREST AND THE HYPERGRAPH;IT DEPICTS THE FOREST BY NESTING. THE LOWER TWO DIAGRAMS REPRESENT THE TWO STRUCTUES SEPERATELY, IN A CONVENTIONAL MANNER. THE HYPERGRAPH OF $\breve{G}$ IS THE ONE ILLUSTRATED IN FIGURE 2-1(MILNER, 2008A).

Interfaces: "*A Bigraph has interfaces, which define its use as a construction block"* (Milner, 2008a).

Consider Figure 2-3 where $\breve{F}$ represents informally a 'portion' of $\breve{G}$ having only some of its nodes. Also one of the hyperlinks is broken.

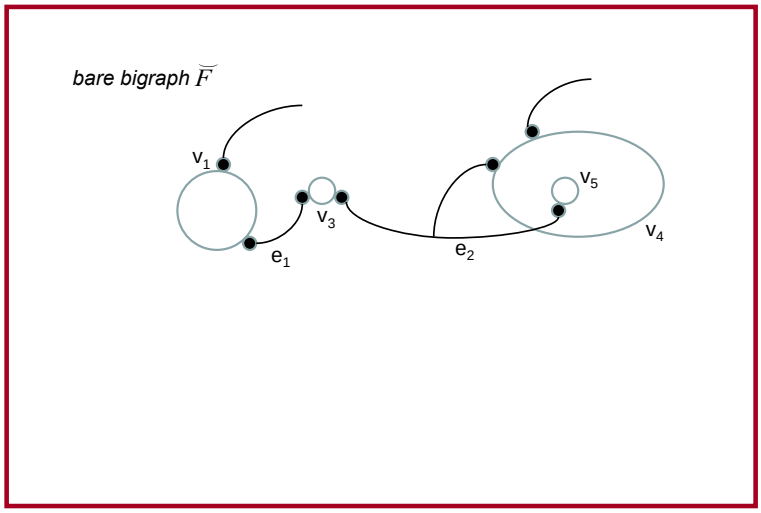

FIGURE 2-3: A BARE BIGRAPH $\breve{F}$ (Milner, 2008a).

To make $\breve{F}$ formally part of $\breve{G}$, *interfaces* are added to bare Bigraphs. This extends $\breve{F}$ and $\breve{G}$ to Bigraphs F and G. Thus, the occurrence of F as a substructure of G can be represented by an equation

$$G = H \circ F.$$

In this equation, H is a 'host' or contextual Bigraph. This extension is done independently for forests and hypergraphs. *"A forest with interfaces will be called a place graph. Similarly, a hypergraph with interfaces will be called a link graph" (Milner, 2008a).*

The interface of a place graph is a finite ordinal $n = \{0, 1, \ldots, n\text{-}1\}$. The members of a place graph's *outer* interface are its *roots*. Similarly, the members of a place graph's *inner* interface are its *sites*.

The outer and inner faces are also called *faces*.

For a link graph, both the outer and inner faces are *name-sets*. The outer face is called the *outer name* and the inner face is called the *inner name*. These names are assumed to be drawn from a countably infinite vocabulary χ.

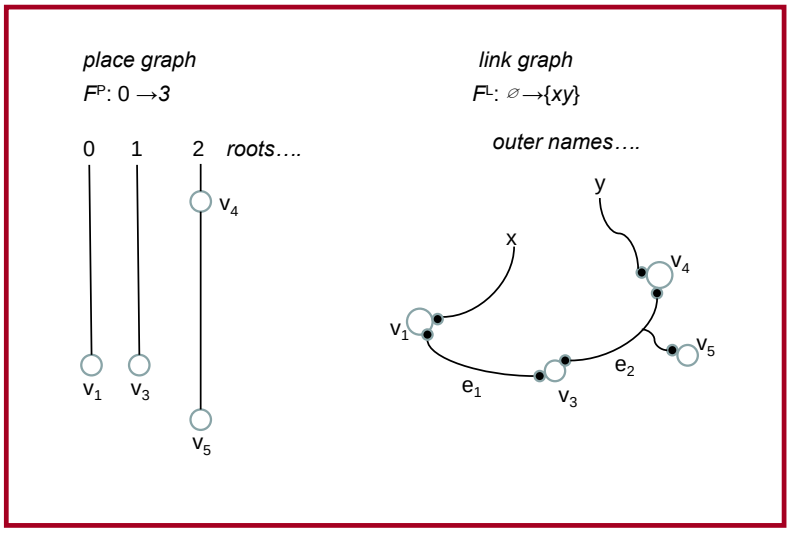

FIGURE 2-4: DEFINING INTERFACES FOR THE PLACE GRAPH AND THE LINK GRAPH OF THE BARE BIGRAPH $\check{F}$ (MILNER, 2008A).

Consider Figure 2-4 which shows how we can define faces for bare bigraph $\check{F}$. Let us choose to have the outer face 3 = {0,1,2} for the forest $\check{F}$. This outer face provides the nodes v_1, v_3 and v_4 with distinct roots as parents. We choose to have no sites and so the inner face is 0.The resulting place graph is written as:

$$F^P: 0 \rightarrow 3$$

This is shown on the left hand side of Figure 2-4. By convention Milner uses $\{xy\ldots\}$ to mean a set $\{x,y,\ldots\}$ of names (Milner, 2008a). We can choose the outer face {xy} to name the parts of the broken hyperlink and inner face ∅ for the hypergraph of $\check{F}$. Then, the resulting link graph is written as:

$$F^L: \emptyset \rightarrow \{xy\}$$

Using the above definitions, a Bigraph is defined as a pair of a place graph and a link graph:

$$B = \langle B^P, B^L \rangle$$

Together, the place graph and link graph are the Bigraph's *constituents*. The outer face of the Bigarph B is a pair $\langle n, Y \rangle$. The first member of this pair n is the outer face of B^P and the second member of the pair Y is the outer face of B^L. The inner face $\langle m, X \rangle$ is defined likewise.

Consider the Bigraph $F = \langle F^P, F^L \rangle$ using the place graph F^P and the link graph F^L discussed earlier. The outer face is $\langle 3, \{xy\} \rangle$. Similarly, the inner face is $\langle 0, \emptyset \rangle$. Now, the trivial interface

$$\epsilon \stackrel{\text{def}}{=} \langle 0, \emptyset \rangle$$

is defined as the *origin*. Thus, we can write the Bigraph F as:

$$\text{F}: \epsilon \rightarrow \langle 3, \{xy\} \rangle$$

The Bigraph F is depicted in Figure 2-5.

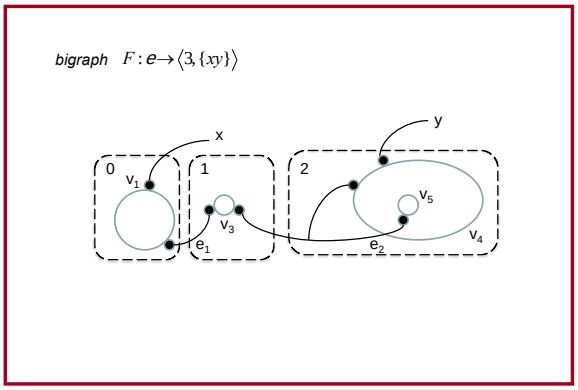

FIGURE 2-5: THE BIGRAPH *F* (MILNER, 2008A).

The dashed rectangles in Figure 2-5 represent *roots*. The rectangles are often referred to as *regions* in Bigraph literature. In the figure the four links belong to the link graph F^L. Out of these four links, two links the edges e_1 and e_2 are called closed links. The remaining two links are named *x* and *y* .The links *x* and *y* are called *open* links.

We can also extend the bare Bigraph of Figure 2-2 $\check{G}$ to a Bigraph G by adding interfaces. All the links in $\check{G}$ are edges because there are no open links. As a result, the name-set in its outer face will be empty. If we give the two nodes v_o and v_4 two roots as parents, then we can place *G* in a larger context because these nodes could be having distinct parents. Bigraph G is shown in Figure 2-6. Because the forest and hypergraph structures are independent, in the upper diagram of that figure, it of no consequence as to where a link crosses the boundary of a node or a region.

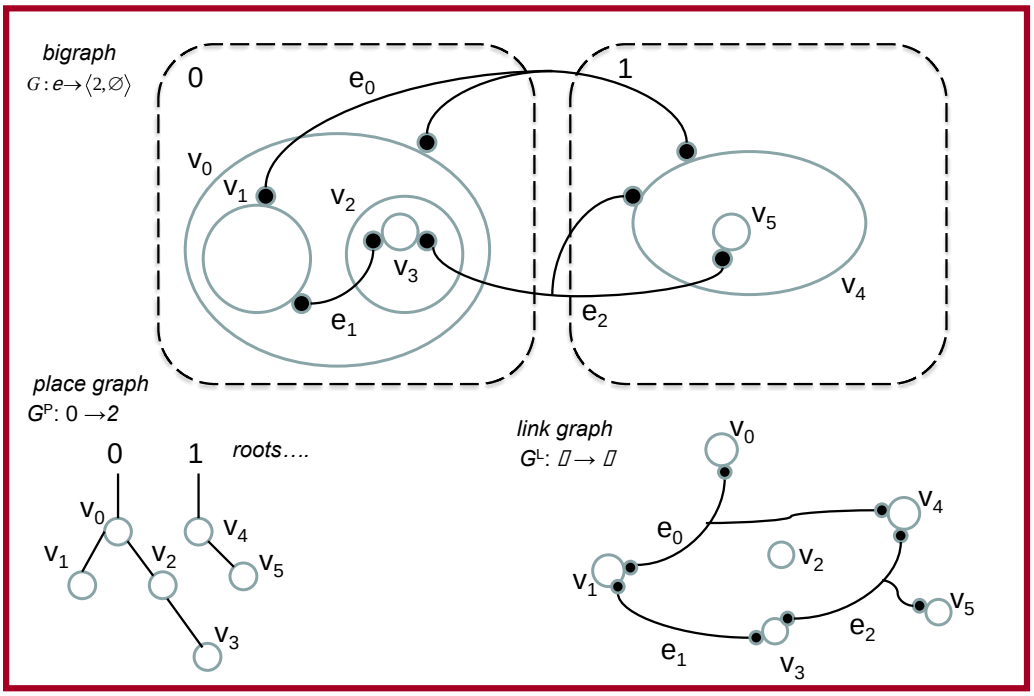

FIGURE 2-6: BIGRAPH G (MILNER, 2008A).

Classification: *"The nodes of a Bigraph may be of different kinds: this reflects that they may contribute differently to dynamics"* (Milner, 2008a).

"Each application of Bigraphs requires a signature" (Milner, 2008a): A basic signature takes the form (κ, ar). It has a set κ whose elements are kinds of nodes called *controls*, and a map $ar: \kappa \to \mathbb{N}$ assigning an *arity*, a natural number to each control. The signature is denoted by κ when the arity is understood. A Bigraph over κ assigns to each node a control whose arity indexes the *ports* of a node where links may be connected.

An application will have different controls which will be specified in a signature. Along with the controls, their arities are also specified. A signature is specified in the following manner:

$$\kappa = \{K:2, L:0, M:1\}$$

A node's arity is the arity of its control in any Bigraph over κ. Often a node's identifier v is omitted in a diagram and its control is shown instead.

The Bigraph $G : e \to \langle 2, \varnothing \rangle$ shown in Figure 2-6 can therefore be depicted as shown in Figure 2-7.

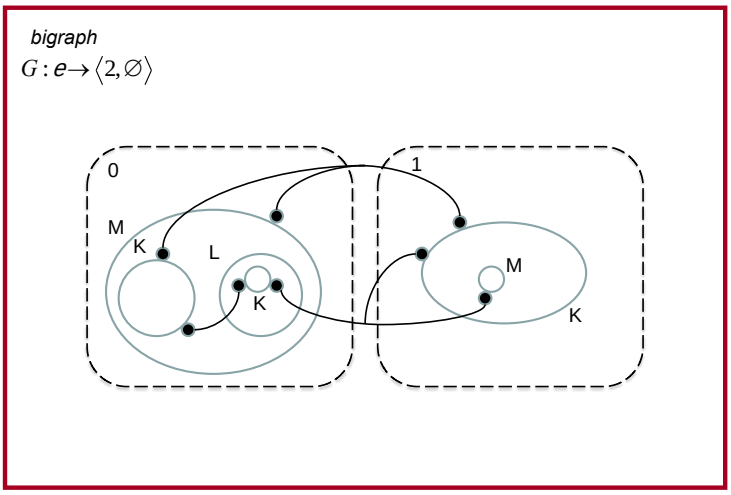

FIGURE 2-7: BIGRAPH G WITH CONTROLS (MILNER, 2008A).

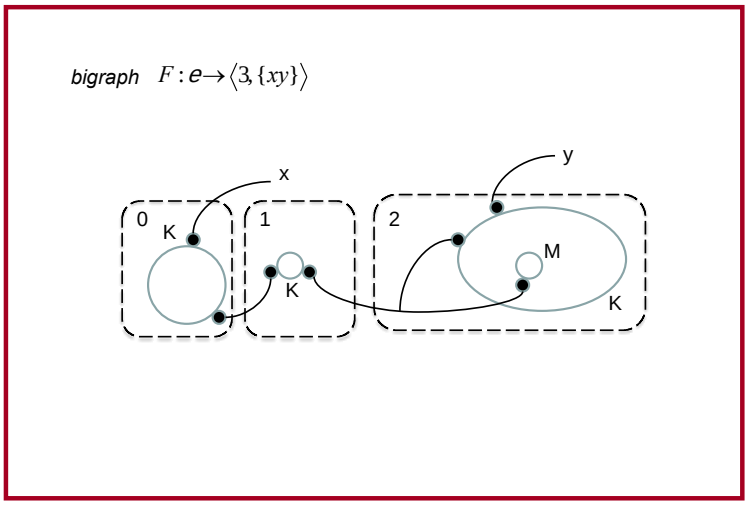

FIGURE 2-8:BIGRAPH F WITH CONTROLS (MILNER, 2008A).

If the node and edge identifiers are present, the Bigraph is called *concrete* otherwise *abstract*. The discussion that follows will deal with abstract Bigraphs unless otherwise specified.

Construction: *"We make larger Bigraphs from smaller ones via their interfaces; this construction is defined in terms of the constituent place and link graphs"* (Milner, 2008a).To construct a Bigraph H such that $G = H \circ F$, the inner face of H must be $\langle 3, \{xy\} \rangle$ which is the same as the outer face of *F*. This means, *H* must have three *sites* 0, 1 and 2. It must also have

two inner names *x* and *y*. In Figure 2-9 we show H and in Figure 2-10 its constituent parts. Its sites are shown as shaded rectangles in Figure 2-9. These two figures illustrate informally the concepts of categorical construction (Milner, 2008a).

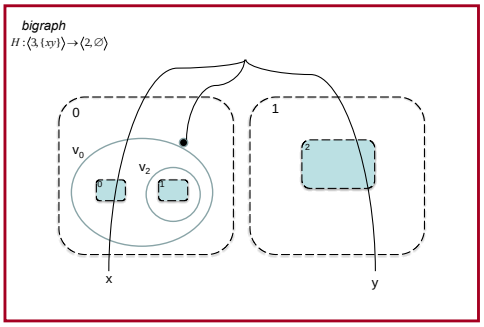

FIGURE 2-9: THE BIGRAPH H (MILNER, 2008A).

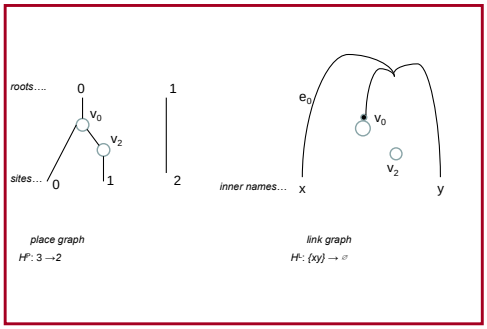

FIGURE 2-10: CONSTITUENT GRAPHS OF THE BIGRAPH H (MILNER, 2008A).

From the place graph drawn in Figure 2-10, we see that each site and node has a parent. This parent is either a node or a root. In Figure 2-9, just as it is of no consequence as to where a link crosses the boundary of a node or a region; it is of no consequence as to where a link crosses a root boundary.

Similarly, from the link graph drawn in Figure 2-10, we see that each inner name and port belongs to a link. This link is either closed or open. A name can be simultaneously inner and outer irrespective of whether they are in the same link.

In Figure 2-11, we show as an abstract Bigraph:

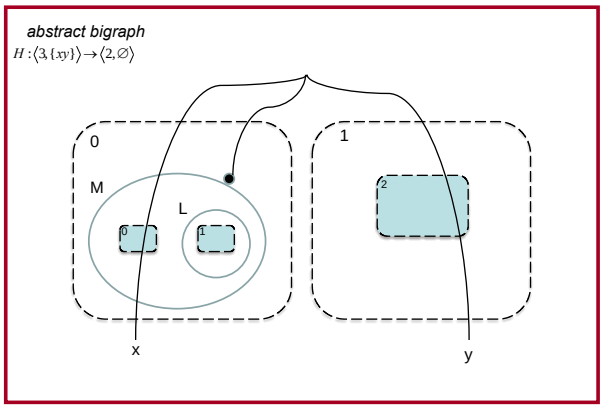

FIGURE 2-11: THE ABSTRACT BIGRAPH H (MILNER, 2008A).

We now present the following informal categorical construction of Bigraphs that follows Milner's description (Milner, 2008a):

Let $F : I \to J$ and $H : J \to K$ be two Bigraphs with disjoint nodes and edges where

$$\mathrm{I} = \langle l, X \rangle,$$

$$\mathrm{J} = \langle m, Y \rangle,$$

$$\text{and } \mathrm{K} = \langle n, Z \rangle.$$

Then the composite H o F : $I \to K$ is just the pair of composites

$$\langle H^p o F^p, H^l o F^l \rangle$$

whose constituents are constructed informally like so:

- To form the place graph $\langle H^P o F^P : l \to n \rangle$, for each $i \in m$ join the ith root of F^p with the ith site of H^p;
- To form the link graph $\langle H^L o F^L : X \to Z \rangle$, for each $y \in Y$ join the link of F^L having the outer name y with the link of H^L having the inner name y.

Therefore H and F are joined at every place or link in their common face J, which ceases to exist.

2.2.1.1.2 ALGEBRA OF BIGRAPHS

"Diagrams are valuable for rapid appreciation of a system's structure. On the other hand algebra is essential, to express and manipulate the ways in which a system may be resolved into components" (Milner, 2008a). We now discuss the algebra that is needed to express the structure of Bigraphs. We will discuss the dynamics of Bigraphs through reaction rules in the next section.

Interfaces: As discussed earlier, an interface is defined as:

$$\mathrm{I} = \langle n, X \rangle$$

If $X = \emptyset$, we abbreviate the interface to $I = n$; if n = 0 we abbreviate it to $I = X$, or $I = x$ if $X = \{x\}$.

On the other hand, if $n = 1$ the interface is called prime. An empty interface of the form

$$\epsilon = \langle 0, \emptyset \rangle$$

is called the *origin*.

The category of Bigraphs (Milner, 2008a): The abstract Bigraphs over a given signature form a category of interfaces I, J,.. as objects and bigraphs $\mathrm{F} : I \to J$ as arrows. If $I = \epsilon$ then F is called *ground*. On the other hand, if J is prime then F is said to be *prime*.

As discussed earlier given $F : I \to J$ and $G : J \to K$, the composite G o F is formed by placing the roots of F in the sites of G and eliding each open link y of F with every link of G that contains the inner name y.

This category has well behaved operation for juxtaposing two disjoint Bigraphs $F_0 : I_0 \to J_0$ and $F_1 : I_1 \to J_1$. Therefore the category is *strict symmetric monoidal* (*ssm*) (Milner, 2008a). This operation is called the *tensor product* and is written as:

$$F_0 \otimes F_1 : I_0 \otimes I_1 \to J_0 \otimes J_1$$

If $I_i = \langle m_i , X_i \rangle$ where (i = 0,1) and X_0 , X_1 are disjoint then

$$I_0 \otimes I_1 \stackrel{\text{def}}{=} \langle m_0 + m_1, X_0 \uplus X_1 \rangle$$

Similarly, we can define $J_0 \otimes J_1$.

Thus, the product $F_0 \otimes F_1$ of F_0 and F_1 is formed by laying them side by side. All algebraic expressions in Bigraphs are defined in terms of product and composition, which enjoy pleasant properties.

Operations: Three operations are derived from the composition and tensor operations discussed above. These are the *parallel product*, the *prime product* and *nesting*.

Parallel and prime products are first defined on arbitrary interface $J_i = \langle n_i, Y_i \rangle$ (i = 0, 1) as follows:

$$J_0 \parallel J_1 \stackrel{\text{def}}{=} \langle n_0 + n_1, Y_0 \cup Y_1 \rangle$$

$$J_0 \mid J_1 \stackrel{\text{def}}{=} \langle 1 , Y_0 \cup Y_1 \rangle$$

Subsequently, the products of Bigraphs F_i: $I_i \to J_i$ (i = 0,1) are defined as follows:

$$Parallel\ product: F_0 \parallel F_1 : I_0 \otimes I_1 \rightarrow J_0 \parallel J_1$$

$$Prime\ product: F_0 | F_1 : I_0 \otimes I_1 \rightarrow J_0 | J_1$$

These are defined exactly like tensor product except that the links of the shared outer names in $Y_0 \cap Y_1$ are coalesced. Also, the prime product has prime outer face.

The third operation is called nesting and is derived from composition. Let $F: I \rightarrow \langle m, X \rangle$ $G: m \rightarrow \langle n, Y \rangle$. The nesting of F within G is defined by:

$$G.F \stackrel{\text{def}}{=} (id_x \parallel G)\ o\ F : I \rightarrow \langle n, X \cup Y \rangle$$

In the equation above, id_x is defined as follows (Milner, 2009): *For each object I , there is an identity arrow :*

$$id_I : I \rightarrow I$$

We just write id when I is understood.

Dynamics (Milner, 2008a): *'Bigraphs can reconfigure themselves according to reaction rules which can be defined arbitrarily'* (Milner, 2008a).

We will first of all discuss a model of built environment where there are agents, buildings, computers and rooms (Milner, 2008a).The four controls namely agents, buildings, computers and rooms are declared in the signature:

{A:2, B:1, C:2, R:0}

Figure 2-12 shows a Bigraph E with this signature. The node-shapes indicate informally the purpose of each port. The Figure 2-12 shows a particular state that may change due to the movement of agents and other movements. The three agents shown are conducting a conference call represented by the open link x. The short links are used to depict that the agents in a room could be logged in. Also, the computers in a building are linked to form a local area network.

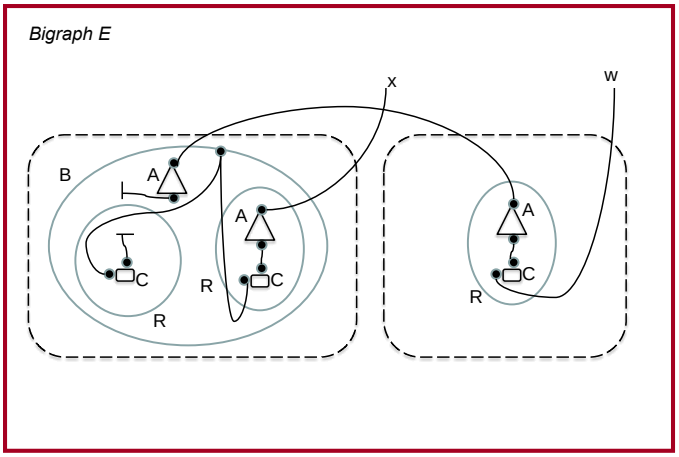

FIGURE 2-12: BIGRAPH E REPRESENTING AGENTS, BUILDINGS, COMPUTERS AND ROOMS(MILNER, 2008A).

Now, to define reconfiguration in the model shown in Figure 2-12, we can specify reaction rules each consisting of a *redex* (the pattern to be changed) and a *reactum* (the changed pattern). Since both these patterns are Bigraphs, they can include both placing and linking. Omitting the precise details of matching, a rule may induce a reaction in a Bigraph *G* if its redex matches a part of *G*. Figures 2-13 to 2-15 show three possible reaction rules.

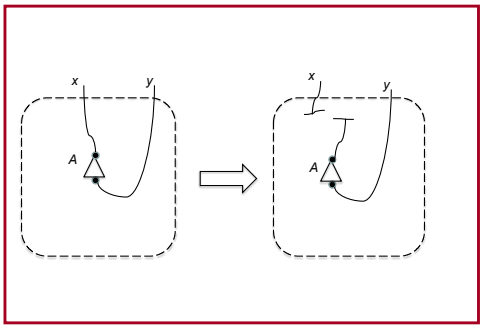

FIGURE 2-13: REACTION RULE B1(MILNER, 2008A).

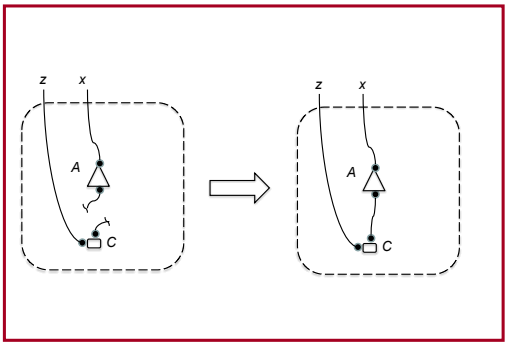

FIGURE 2-14: REACTION RULE B2(MILNER, 2008A).

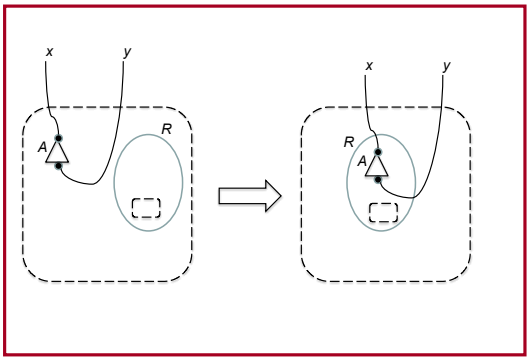

FIGURE 2-15: REACTION RULE B3(MILNER, 2008A).

Figure 2-13 shows reaction rule B1where an agent can leave a conference call. In the left hand side is the redex . This redex can match any agent. The links that point out mean that she may

be linked through her ports to zero or more other ports. These ports can be in the same place or anywhere else. If the link *x* represents a conference call with other agents in other buildings, reaction rule B1 will unlink her. Notice however that the link *y* to the computer is retained.

Figure 2-14 shows rule B2 where the reaction rule's application results in an agent connecting to the computer in the same place. The redex of the reaction rule ensures that a matching will occur only if the agent is not connected to any computer and the computer is not connected to any agent.

Both the reaction rules B1 and B2 just change the linking and not the placing.

Finally, Figure 2-15 depicts reaction rule B3 where the placing has been changed because an agent enters a room. The redex of the reaction rule ensures that a matching will occur only if the agent and the room are co-located- for example in a building. The dashed rectangle depicts a site, which represents a *parameter* of the rule. This allows the redex to match to any room Bigraph that contains other Bigraphs representing say computers. The occupants are allowed to be linked anywhere whether to each other or to nodes that are out side the room by the matching discipline. Moreover, the redex allows the agent's ports to be connected to other nodes. The reactum of the rule retains such connections. The redex also allows the agent to have no link and the context in which the rule is applied may close it off. Consider Bigraph *E* of Figure 2-12. Reaction rule B3 can be applied to Bigraph *E* thereby allowing an agent in the left-hand building to enter a room.

If the reaction rules B1, B2, B3 are applied once to Bigraph E of Figure 2-12, we will get the Bigaph *E'* shown in Figure 2-16.

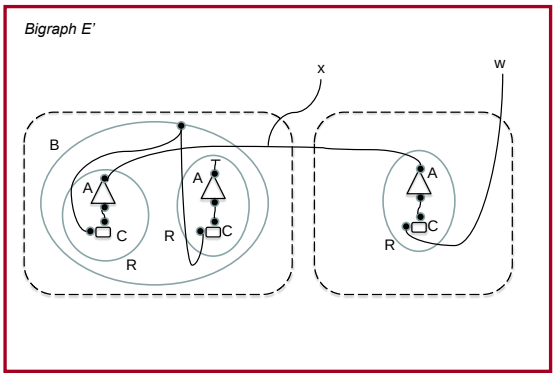

FIGURE 2-16: THE BIGRAPH E' AFTER THE APPLICATION OF REACTION RULES B1,B2,B3 TO BIGRAPH E (MILNER, 2008A).

2.2.1.2 THE BIGRAPH PROGRAMMING LANGUAGE (BPL) TOOL

We have used MiniML syntax-a subset of SML constructs- supported by Bigraph Programming Language (BPL) Tool (ITU, 2011), to implement our system. We now give a brief overview of the BPL Tool Architecture focusing on those parts that we have used in our implementation.

The BPL Tool architecture consists of the following components (ITU, 2007a):

1) Bigraph Programming Language (BPL): This is a *"high level bigraph, signature, and bigraph reaction rule definition language for binding Bigraphs"* (ITU, 2007a). A binding Bigraph is defined as a Bigraph that *"adds lexical scopes on links which locates some names at nodes"* (Elsborg, 2009).Calculi like π - calculus and λ - calculus can be encoded using binding Bigraphs. Note that because we do not use links in our models, we have not needed to utilize the added structure provided by the BPL Tool's implementation of binding.

2) Bigraph Term Language (BGTerm): This *"is a low level term language for Bigraphs closely based on elementary Bigraphs and combinators"* (ITU, 2007a). Because the terms do not need to be well-formed, BGTerm is an un-checked term language. An abstract data type for BGTerm is implemented by the BGTerm module of the BPL Tool including ML constructors.

3) Bigraph Term Language Values (BGVal): These are BGTerms that have been *"checked for well-formedness with interface data"* (ITU, 2007a). An abstract data type for BGVal is implemented by the BGVal module of the BPL Tool including ML constructors. This facilitates the domain specific usage of BPL Tool for example in Plato-Graphic models. The module also provides a total constructor function and special BGVal pattern matching functions for partial deconstruction. We use BPL Tool's SUGAR module (discussed later) for creating BGVals in SML. This is a subset of SML and is called MiniML.

4) Binding Discrete Normal Form (BDNF): These are *"binding Bigraph terms expressed in one of the four forms defined by Damgaard and Birkedal"* (Damgaard and Birkedal, 2006), (ITU, 2007a). An abstract data type for BDNF is implemented by the BgBDNF module of the BPL Tool including ML constructors and de-constructors.

In Figure 2-17, we show the four layers of the BPL Tool that we have just discussed (ITU, 2007a). It shows how an input of a text file is translated as the data flows between the four layers. This data is then input to the rewrite engine. One step of the rewrite engine results in a BGVal output. This needs to be renormalized and then rewriting can continue. The portion of the figure on the left hand side gives details of the Plato-graphic Location Model which is a domain-specific usage of the BPL Tool code.

The Figure 2-18 (ITU, 2007a), shows the details of the rewrite engine of Figure 2-17. In this example of the re-writing process, a user-driven scheduler accepts an agent in the BDNF form as an input, a signature on BDNF-level form, and a set of rules which are also in BDNF-level form. In step 1, the user can choose a rule through a user interface provided by the BPL-Tool (user interface not used by us in this thesis though) to find all possible reactions for a specific rule. Another possibility is finding just some reaction. Alternatively, in step 2, the user can choose to find all reactions for all rules. Once an action is chosen, the "Matcher" is invoked. This invocation can be with either a single rule or all the set of rules. A set of matches is returned. The user can invoke the rewriter to perform a specific reaction provided there is at lest a single match. The rewrite is performed by the "Rewriter" by substituting the reactum for the redex and also instantiating the parameters. The agent needs to be re-normalized in general after a rewrite step.

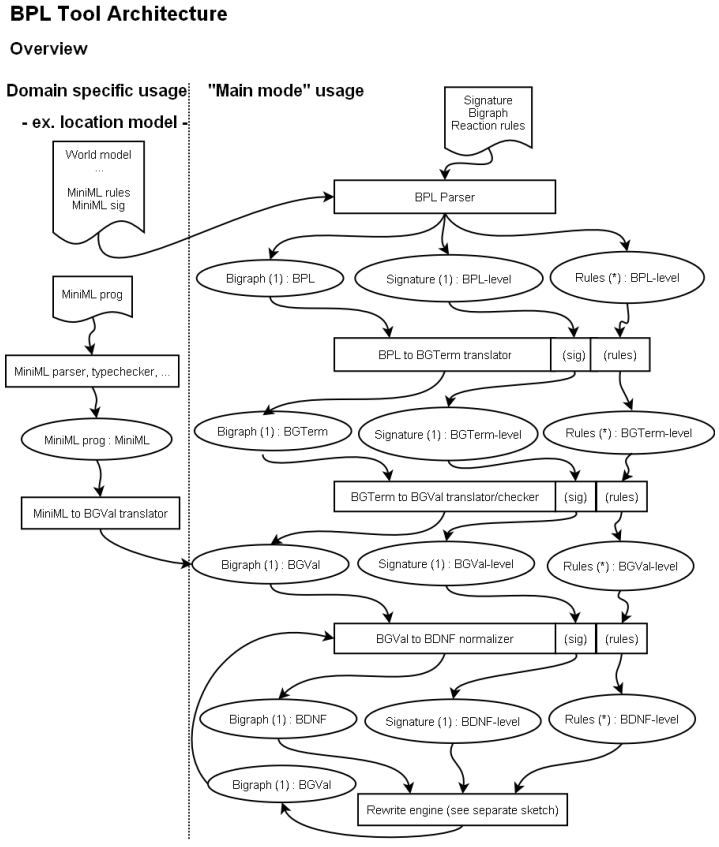

FIGURE 2-17:ARCHITECTURE OF THE FOUR LAYERS OF BPL TOOL(ITU, 2007A).

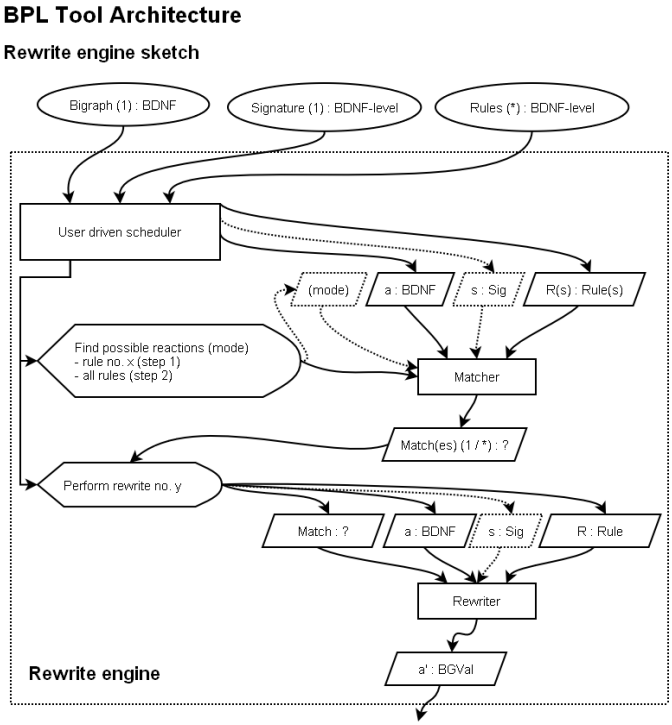

FIGURE 2-18: THE REWRITE ENGINE OF THE BPL TOOL (ITU, 2007A).

As discussed earlier, the SUGAR module is used to enter the Bigraph values directly into SML syntax. The module defines syntactic sugar for creating BGVals in SML (ITU, 2007b). This subset of SML is called MiniML. We use MiniML to program our implementation. Implementing with MiniML enables us to use SML's control constructs, which are lacking in Bigraphs. This access to SML's control constructs is one of the reasons why MiniML was developed (Elsborg, 2009). We will give detailed explanation of how to program with MiniML in Chapters 3 and 4.

2.2.1.3 USING THE SUGAR MODULE OF THE BPL TOOL

We present a short survey of the features of Bigraphs that we have used for expressing a model at runtime with the BPL Tool (ITU, 2011). We use the MiniML syntax which is a subset of the Standard ML (SML) constructs, provided by the BPL Tool, to represent Bigraph expressions. In particular, we will use BPL Tool's SUGAR auxiliary module which defines the syntactic sugar for entering Bigraph expressions directly in SML.

a) Placing and Linking: A Bigraph consists of two independent structures- a place graph and a link graph. These two graphs share nodes. The place graph is restricted to be a tree (See Figure 2-19 where a device node is nested inside a shopping mall node). A link graph can be a hyper-graph i.e. a link can connect more than two objects (ITU, 2008). We have not used link graphs in our implementation using the BPL Tool (ITU, 2011). This is because the matching algorithm of the BPL Tool is not designed to efficiently handle a huge explosion of links that occurs as the

size of the Bigraph grows (Elsborg, 2009). Leaving the links out is the result of the in-efficient implementation of the BPL Tool rather than any inherent limitation of Bigraph Theory (See Chapter 5). In what follows therefore, we will only talk about place graphs.

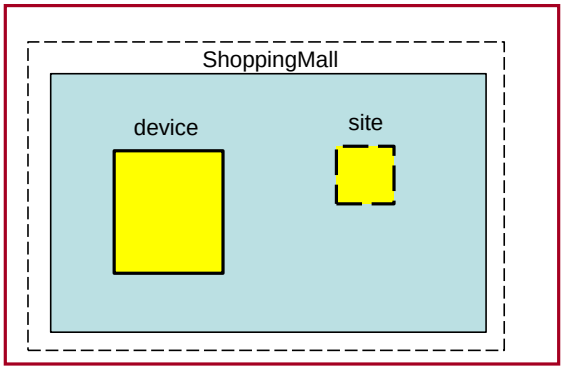

FIGURE 2-19: A PLACE GRAPH WITH A ROOT AND A SITE.

b) Interfaces: As discussed earlier, these make parts of a Bigraph externally accessible and define its use as a construction block (ITU, 2008),(Milner, 2008a). In the Figure 2-19, a place graph is shown with the nodes of the tree nested inside each other. There are two controls for the two nodes: a control called `ShoppingMall` and another called `device`. The outermost dashed rectangle represents the root and the innermost dashed rectangle represents a site. A place graph's outer interface is its root and inner interface its site. In our figures, we omit the drawing of the root node when there is no ambiguity.

c) Classification of nodes: As discussed earlier, in the place graph, each node has a *control* which is the name of the type of node. Each control has a *status* which could be *atomic, active or passive.*

Atomic status of a node: Atomic nodes are those nodes that cannot have child nodes or *sites* (holes within which other nodes can fit in) and no *reaction rules* (See point (f) Dynamics on the following pages) are allowed inside,

Active status of a node: Active nodes are those nodes that can have child nodes or sites and reaction rules are allowed inside,

Passive status of a node: Passive nodes are those nodes that can contain child nodes but reaction rules are allowed only at the node and not in their child nodes.

We construct atomic, active and passive nodes in Chapter 4.

The parent and child nodes can be of different kinds (i.e. different controls) - for example, a device node could be inside a location node (Figure 2-19).

d) As discussed earlier, the signature of a Bigraph declares its types of controls, their status, and their arity.

e) Operations to construct larger Bigraphs from smaller ones: The three operations in MiniML syntax ('S' is the SUGAR module of BPL Tool) that we use are:

• Composition: - S.o (G, F). This means put Bigraph F inside Bigraph G provided G is not an atomic node and the number of sites in G are equal to number of roots in F. If F denotes a device of control `device` with a root and G denotes a shopping mall with a site then we get a Bigraph shown in Figure 2-20 by putting `device` inside `ShoppingMall` Bigraph's site. Notice that we have omitted `ShoppingMall` Bigraph's root to avoid clutter:

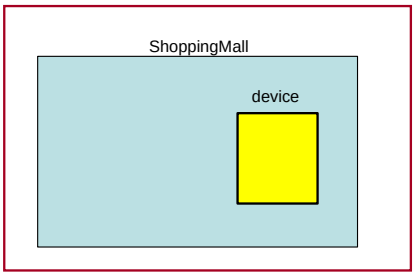

FIGURE 2-20: COMPOSITION OPERATION OF BIGRAPHS.

• Parallel Product: - S.||(G,F). This means place Bigraphs G and F side-by-side. If G denotes say a Bigraph of a control called WORLD and F denotes a Bigraph of a control called SCA (Service Component Architecture), then we get the Bigraph shown in Figure 2-21. Assume for now that WORLD and SCA are just arbitrary names that we give to these Bigraphs. We will discuss what WORLD and SCA stand for in Chapter 4.

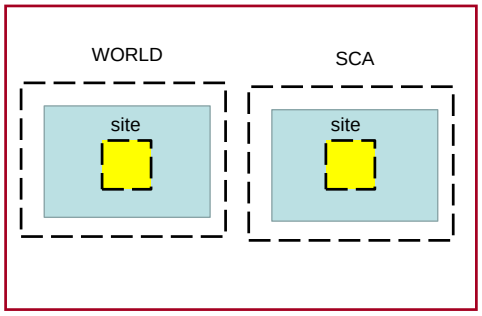

FIGURE 2-21: PARALLEL PRODUCT OPERATION OF BIGRAPHS.

• Prime Product (also called Merge Product): - S.`|`(G,F). This means, place Bigraphs G and F side-by-side under a common parent node. If G denotes say a Bigraph of a control called WORLD and F denotes say a Bigraph of a control called SCA and we wish to place both side-by-side, then in MiniML syntax, we write S.`|`(WORLD, SCA). This is depicted in Figure 2-22. Notice that the two Bigraphs WORLD and SCA are under a common root after the prime product operation. This common root is formed by coalescing the two individual roots of the two Bigraphs WORLD and SCA.

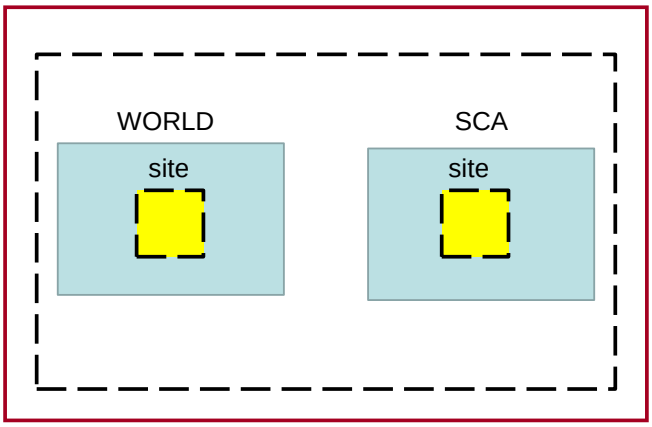

FIGURE 2-22: PRIME PRODUCT OPERATION OF BIGRAPHS.

f) Dynamics: The reconfigurations of the structure of Bigraphs can be specified by defining reaction rules. A reaction rule consists of a redex which is a pattern to be changed and a reactum which is the changed pattern. Reaction rules can be parametric if both the redex and reactum have sites. These sites are the parameters of the reaction rules and can be considered ‘don’t care’ when the reaction rule is applied. Each site in the reactum must be mapped to a site in the redex. In Figure 2-23, we depict the reaction rule ‘Device un-cached’ where the Bigraph with control `device` is the node representing a device and the Bigraph with control `location` is the node representing the location. The site is a ‘don’t care’ as it is a parameter for the reaction rule and could contain any Bigraph.

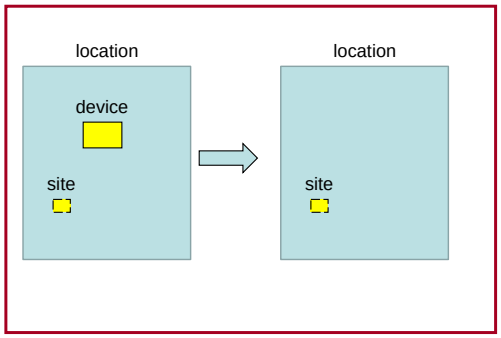

FIGURE 2-23: DEVICE 'UNCACHED' RULE.

A matching algorithm is used by the BPL Tool to figure out where in a large Bigraph the redex of the reaction rule should be applied (see Section 5.3 for an example). Matching is NP-Complete (Højsgaard, 2011). Modelling through reaction rules enables us to handle run-time complexity. This is because a few rules can intensionally construct the set of infinitely many possible re-configurations of the system. An intensional construction of a set defines the basis, inductive rules and a closure property to generate a set. In contrast an extensional construction enumerates all the elements of a set.

Thus, a Bigraph's syntax can be used to define processes and its reaction rules can be used to define how those processes interact. The syntax and the reaction rules are together called Bigraphical Reactive Systems (BRS) (Milner, 2009), (Birkedal et al., 2006).

2.2.1.4 USING THE SUGAR MODULE TO DESCRIBE PLATO-GRAPHIC MODELS

Because Bigraphs lack control structures, reaction rules that model physical action cannot be used to compute directly with a model of that action. For example to implement recursive queries, we will need to encode in Bigraphs a runtime stack with additional controls (Elsborg, 2009).This problem is addressed by Birkedal et al. (Birkedal et al., 2006) by representing three separate concerns in three Bigraphical Reactive Systems (BRS). These three BRSs constitute the Plato-Graphic model (PGM) as defined by Birkedal et al (Birkedal et al., 2006). We now discuss some definitions that are used to describe the Plato-graphical model. We have taken these definitions from Elsborg's thesis and for a thorough introduction, we refer the readers to that thesis (Elsborg, 2009).

Notation 1 (Elsborg, 2009): *"We write* $B = (\kappa, \mathcal{R})$ *and* $B' = (\kappa', \mathcal{R}')$ *to indicate that B is a Bigraphical Reactive System with controls* κ *and rules* $\mathcal{R}$*, and write* $f \in B$ *to mean that f is a Bigraph of B."*

Definition (Independence) (Elsborg, 2009): *"Let $B = (\kappa, \mathcal{R})$ and $B' = (\kappa', \mathcal{R}')$ be bigraphical reactive systems. Say that B and B' are independent and write $B \perp B'$ iff κ and κ' are disjoint."*

Definition (Composite Bigraphical System) (Elsborg, 2009): *"Let $B = (\kappa, \mathcal{R})$ and $B' = (\kappa', \mathcal{R}')$ be Bigraphical reactive systems. Define the union $B \cup B'$ point-wise, i.e., $B \cup B' = (\kappa \cup \kappa', \mathcal{R} \cup \mathcal{R}')$, when κ and κ' agree on the arities of the controls in $\kappa \cap \kappa'$."*

Definition (Plato-graphical model) (Elsborg, 2009): *"A Plato-graphical model is a triple $(\boldsymbol{C},\boldsymbol{P},\boldsymbol{A})$ of Bigraphical reactive systems, such that $\boldsymbol{M} = \boldsymbol{C} \cup \boldsymbol{P} \cup \boldsymbol{A}$ is itself a Bigraphical reactive system and $\boldsymbol{C} \perp \boldsymbol{A}$. A state of the model is a Bigraph of M on the form $/\vec{x}.(C \parallel P \parallel A)$ where $C \in \boldsymbol{C}$, $P \in \boldsymbol{P}$, and $A \in \boldsymbol{A}$, and $\vec{x}$ is some vector of names."*

Of the three, the first BRS 'World' (W) models the environment. The second 'Proxy' (P) models the information about the World (W). The third 'Agent' (A) models an application that queries the Proxy (P) about the World (W). See Figure 2-24.

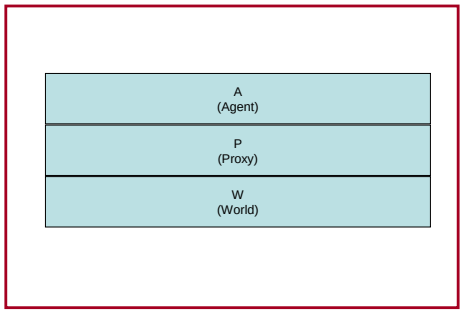

FIGURE 2-24: THE PLATOGRAPHIC MODEL.

Reaction rules are used to represent the 1) Dynamics of the real world in the World BRS and to 2) Model queries in the Proxy BRS.

The constraint on PGMs is that there are no common controls (types of Bigraphs) between the 'World' BRS and 'Agent' BRS. A reaction rule can involve Bigraphs in more than one layer simultaneously. However, reaction rules spanning the 'Agent' BRS and the 'World' BRS are also not allowed.

2.2.2 WHAT IS BEING MODELLED WITH BIGRAPHS

We now discuss the practical application of Bigraphs. Since Bigraphs capture *"discrete space involving adjacency and containment"* (Milner, 2009) through representation of locality and connectivity, it is perhaps apt that to date most of the applications have used Bigraphs to

represent architecture of software systems in the following not necessarily mutually exclusive categories:

1) Architectures of volatile systems.

2) Architectures where services are the main communicating entities.

3) Architectures for software systems in domains other than volatile systems.

We analyze the papers in each category now.

2.2.2.1 EXPRESSING ARCHITECTURES OF VOLATILE SYSTEMS

Volatile systems encompass mobile and hand-held computing systems, ubiquitous computing systems, wearable computing systems, context-aware computing systems, tangible computing systems and augmented reality systems (Coulouris, 2012). Bigraphs have been used as a language to express the architecture and reconfiguration of a volatile system. We have already discussed Birkedal et al.'s idea (Birkedal et al., 2006) of a Plato-graphic model for context aware systems in section 2.2.1.4. In chapter 4, we discuss how we have used their ideas in the design of our own model at runtime. The following papers have expressed the architecture of a volatile system using Bigraphs:

1) Chris Greenhalgh and co-workers have proposed the Bigraphspace library (Greenhalgh, 2009a, Greenhalgh et al., 2009, Greenhalgh, 2009b) as a *"shared distributed data structure that can be used for communication and coordination between components in a ubiquitous software system"*. This is similar to the idea of a tuple-space (Gelernter, 1985). Bigraphspace represents Bigraphs as a Document Object Model (DOM) tree. The XML element hierarchy models the Bigraph's place graphs. Cross-references between XML elements model the Bigraph's link graphs. A client of Bigraphspace can query it using a Bigraph pattern (redex) that is then matched with the Bigraphspace's Bigraphical structure. This Bigraphical structure can be updated by using reaction rules to reflect changes in the real world. Bigraphspace is intended as a foundational layer on top of which a suite of supporting modeling/authoring/software development tools is proposed to be developed. Also, Bigraphspace is intended to support a runtime system by maintaining an up-to date model reflecting changes in the environment.

The idea of representing tuple-space like structure with Bigraphs is an innovative contribution towards exploring practical application of Bigraphs.

2) The paper by Walton and Worboys (Walton and Worboys, 2009) uses Bigraphs to model topological and physical image schemas of built environments. Built environments are volatile in that the structures within them are changing constantly. They define image schemas as

"abstractions of spatio-temporal perceptual patterns". Image schemas do not have widely accepted formalisms to represent them. Two image schemas are modelled: the container image schema represented by one Bigraph contained inside another and the link image schema where two Bigraphs are connected by a link. Two schemas are composed to construct a larger Bigraph. Consider Figure 2-25 showing Bigraph I. The rectangle marked 1 represents an open space where an agent A is shown linked to the key K. K's open link x_1 represents the fact that the key's lock is unknown. The rectangle marked 2 represents another open space where a lock represented by the rectangle L guards another open space represented by rectangle 3. Lock L's key is unknown and this is represented by the open link x_2.

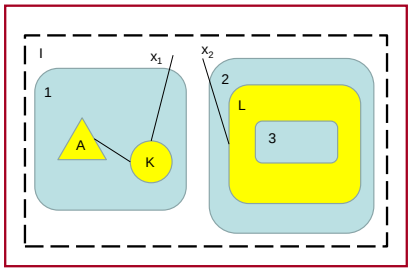

FIGURE 2-25: BIGRAPH I (WALTON AND WORBOYS, 2009).

Reaction rules model the moving in and out of a Bigraph that represents a container as well as the establishment and breakage of links between Bigraphs. More complex image schemas are modelled by firing a series of reaction rules one after another.

There is a rich theory of image schemas (Johnson, 1987) that the authors have tapped in to pick out those schemas that they model with Bigraphs. This theory enables them to capture all the necessary elements of physical and virtual spaces in their Bigraph models. Their proposed model is meant to be used to aid navigational tasks of agents in physical and virtual spaces.

3) The paper by Xu et al. (Xu et al., 2011) models context aware mobile systems with Bigraphs and illustrates their ideas by applying them to a smart phone example. They model context using the structure of Bigraphs and changes in context by using the reaction rules of Bigraphs.

Although their application of Bigraphs is interesting, they have not gone much beyond previous work by Birkedal et al.(Birkedal et al., 2006), Milner (Milner, 2009),(Milner, 2008a), Debois and Damgaard (Debois and Damgaard, 2005) and Greenhalgh (Greenhalgh, 2009a).

4) Wang et al.'s (Wang et al., 2011) paper has two authors in common with Xu et al.'s (Xu et al., 2011) paper above. Similar to Xu et al., Wang et al. factor out common elements of a context aware system from various definitions of context awareness. These common factors are

then modelled using abstractions of Bigraphs. They present an example of a context aware hospital and an example of a university project being managed in a context aware space.

As with Xu et al.'s paper above, this paper has used Bigraphs in interesting domains which is conceptually similar to the ideas presented in Birkedal et al.(Birkedal et al., 2006), Milner (Milner, 2009),(Milner, 2008a), Debois and Damgaard (Debois and Damgaard, 2005) and Greenhalgh (Greenhalgh, 2009a).

5) Zhai et al.'s paper (Zhai et al., 2011b) uses Bigraphs to model passengers getting a ticket, entering a metro station and leaving it-each of which is a Bigraphical reaction rule. The nodes in the Bigraph model a person, a gate, a ticket and the rest of the metro system. In future work, Zhai et al. plan to use a more complete Bigraphical model of the metro system for running a simulation to predict passenger volume, train dispatch, income distribution etc.

The ideas presented in the paper are interesting in that in that they consider passengers as part of their model of the metro system and its embedded devices.

6) Another paper (Zhai et al., 2011a) by the same lead authors as above and in the same domain of an urban metro rail system uses a different set of reaction rules to model slightly different activities- namely those of getting onto a train, transferring en-route to a different line owned by another operator, and getting off a train. The problem that they seek to address is how to share passenger fares between different operators of different lines in the barrier free transfer mode of an urban metro rail system. They present an algorithm that calculates time cost of all possible paths between two stations. Each path's time cost is the sum of the time costs of all the transfer reaction rules that need to be triggered. The path that takes the shortest time is predicted to be the path that the passengers will take. They confirm this fact by analyzing the choice of the actual path taken by 500 passengers. They propose that the cost of the ticket can be shared between different operators based on the way this optimal path that the passengers take (and predicted by their algorithm) is divided between the lines owned by different operators.

This paper demonstrates a novel usage of Bigraphs. However, much work needs to be done to consider more complex scenarios of passenger movement when designing their cost prediction algorithm.

7) High Confidence Petroleum extraction Software Systems (HCPESS) are used in petroleum well sites to process information from sensors and actuators embedded within petroleum production equipment. These sensors and actuators are failing in the harsh environment, and thus creating volatility. The paper by Zhai et al. (Zhai et al., 2011c) partially models HCPESS software framework with Bigraphs. The architecture of the HCPESS framework is modelled by

the place and link graph of Bigraphs. The dynamic reconfigurations of the framework are modelled by Bigraphical reaction rules.

Even though HCPESS is an interesting domain for modelling with Bigraphs, more work needs to be done to represent a realistically complex HCPESS framework with Bigraphs.

8) The paper by Henson et al. (Henson et al., 2012) discusses the appropriateness of Bigraphical abstractions for applications in intelligent environments. They hope to use these abstractions to document, design, and analyze such applications. Bigraph nodes model physical spaces, users, devices and Bigraph links model the connections between these entities. The reaction rules model a user entering a physical space, users accessing a physical space using a real or virtual key, a user getting a key, a user using the same key for multiple uses, a key that is shared among users being used multiple times and finally, an establishment of a connection between a device and a physical space's intranet. Their example scenario involves modeling of teachers who can give commands to pupil's devices (Figure 2-26). In Figure 2-26, 'm' nodes represent people and can be assigned the role of teachers or pupils. For instance, the node 'm' situated inside the left-hand side oval space (representing a classroom) and connected to the 'cmd' node (command) represents a teacher. The 'cmd' node is further connected to another 'm' node representing a pupil and situated inside the right-hand side oval space that represents another classroom. The 'cmd' node represents a command from the teacher to disconnect the pupil from their device.

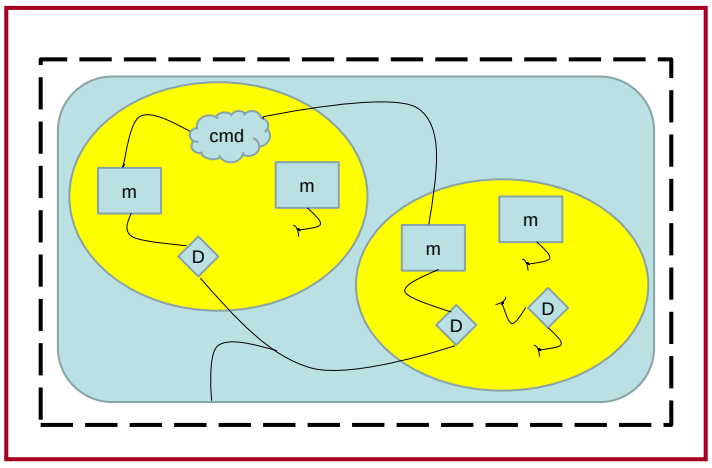

FIGURE 2-26: THE TEACHER/PUPIL RELATIONSHIP (HENSON ET AL., 2012).

They consider Bigraphs as an appropriate abstraction for the following four reasons:

"a) They are intuitive and lie close to the topic of investigation; b) *They are relatively simple to understand and deploy (in contrast to the systems they may analyze)*; c) *They offer a means to*

tame complexity through multiple descriptions at different levels of abstraction; d*) The system itself can be usefully used without having to engage with its mathematical foundations"*.

This paper constitutes an important step towards evaluating the appropriateness of Bigraphical abstractions to model intelligent environments. The authors point out that a reaction rule can match a given Bigraph in several ways. However they do not present a way to deal with cases where only a specific match corresponding to a specific node in the place graph needs to be returned.

9) Another paper by Walton and Worboys (Walton and Worboys, 2012) uses a Bigraph based model to aid agents in navigating indoor spaces for accomplishing a goal. The work presented in this paper complements the work in the paper discussed above. The structure of the Bigraphs is used to model the following:

a) Location of agents and objects.

b) Topological configurations such as building hierarchies.

c) Path based navigation graphs and other non-spatial relations (someone's office).

Reaction rules of Bigraphs model changes in context or effect of an agent's actions. They show how Bigraphs can be used to model indoor spaces usefully even in light of missing information.

Walton and Worboys' paper uses Bigraphs as an aid in goal-directed navigation. The paper presents an excellent discussion on those aspects of indoor spaces that should be expressed in a model.

10) Pereira et al. (Pereira et al., 2012) have proposed a two layered model, similar to our work, to simulate a volatile system. One of the layers models the physical (execution machines and their environments) with Bigraphs and the other models the virtual (software agents) with algebraic structure (See Figure 2-27). The agents are hosted at the nodes of Bigraphs and can interact with them by observing them, migrating to them and controlling them. Agents themselves are sequential computations operating concurrently over the same physical structure and can be represented as finite or infinite state machines. In common with other hybrid systems like Alur and Dill's timed automata (Alur and Dill, 1994) or Henzinger's hybrid automata, Pereira et al.'s model has a hybrid notion of time remembering that a *"hybrid system is a discrete system that interacts with a continuously evolving one namely its environment"* (Aceto et al., 2007). They hope to reuse pi-calculus (Milner, 1999) or SHIFT (Deshpande et al., 1998) to express their agents.

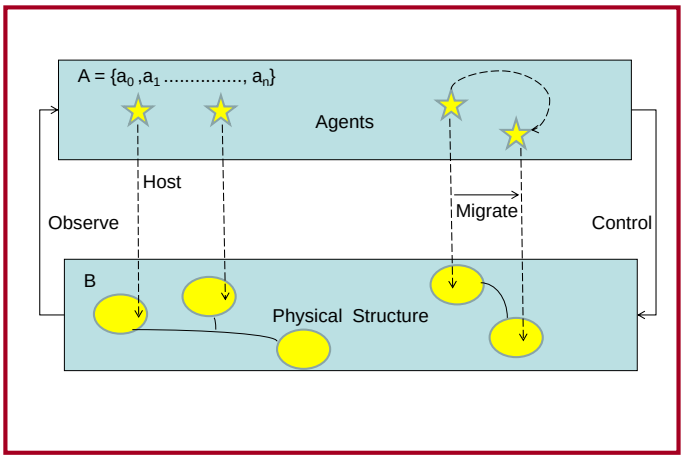

FIGURE 2-27:BLOCK DIAGRAM DEPICTING THE CONTROL LOOP BETWEEN AGENTS AND THE PHYSICAL STRUCTURE (PEREIRA ET AL., 2012).

Pereira et al.'s work is an important contribution to modeling of the separation of concerns by having two layers for two concerns. However, they do not use Bigraphs for expressing both the layers of their model. Expressing both the layers in such a manner would have made their models more expressive as Bigraphs can be looked upon as a meta-language (Birkedal et al., 2006) encompassing Petri nets (Milner, 2004a), pi-calculus (Jensen, 2006), (Jensen and Milner, 2003), (Jensen and Milner, 2004), mobile ambient (Jensen, 2006) and lambda calculus (Milner, 2004b). Also, Pereira et al.'s model is suitable for *simulation* of physical and virtual movement.

2.2.2.2 EXPRESSING ARCHITECTURES WHERE SERVICES ARE THE MAIN COMMUNICATING ENTITIES

Services are programming abstractions with well defined interfaces and constitute the communicating entities in many distributed system architectures (Coulouris, 2012). Such architectures have been modelled using Bigraphs in the following papers:

1) Zhang et al.'s paper (Zhang et al., 2008) is a first attempt towards providing a uniform framework based on Bigraphs to represent service compositions in BPEL-like languages. They use Bigraphs to express BPEL-like language and use this Bigraphical representation to prove properties about the language. They have specified communication, scope-based compensation and error handling of a BPEL-like language with Bigraphs.

This paper represents an interesting application of the specification capabilities of Bigraphs. However, they have not utilized the reaction rules of Bigraphs in their framework.

2) Xue et al. (Xue et al., 2009) use CCS encoded in Bigraphs (Milner, 2009) to model service interaction patterns (Barros et al., 2005). Reconfiguration of these interaction patterns are modelled with Bigraph reaction rules. The service interactions modelled in this manner can be used to simulate business collaborations and process choreographies. Moreover, these interactions can be expressed in terms of Bigraphical compositions.

The modeling of service interactions is important to capture all possible interaction patterns of business processes.

3) The paper by Huai-Guang et al. (Huai-Guang et al., 2010) seeks to explore ways in which service compositions could be reconfigured dynamically and be guaranteed to run correctly. The structure of a composition is modelled using the place graph and link graph of Bigraphs. Six Bigraphical reaction rules model the service interaction patterns in the composition. An example case study modeling a user requesting for airline and hotel reservation through their mobile devices is also discussed.

The difference between this paper and the paper by Xue et al.(Xue et al., 2009) discussed above is that in the former Bigraphs are used directly to model service interactions whereas in the latter, CCS expressed in Bigraphs is used to model service interaction patterns.

This paper's idea of modelling service interaction with reaction rules is innovative. However, the example case study describes composition of web services rather than the composition of simpler services running on ubiquitous computing devices.

2.2.2.3 EXPRESSING ARCHITECTURES FOR SOFTWARE SYSTEMS IN OTHER DOMAINS

We now discuss the papers that have used Bigraphs to express architectures for software systems other than volatile systems.

1) Debois and Damgaard's technical report (Debois and Damgaard, 2005) uses Bigraphs to model an internal switch, finite automata, the game of "life" (Gardner, 1970), combinatorial logic, term unification and an event driven system. These systems' structure is modelled by Bigraph place and link graphs. The dynamics of these systems are described by appropriate reaction rules.

This technical report gives a good introduction to modelling with Bigraphs for someone new to such modelling.

2) The paper by Chang et al. (Chang et al., 2007) uses Bigraphs to express two architectural patterns: the client-server pattern for distributed systems and the pipe-filter pattern for a generic

software systems. Reaction rules of Bigraphs are used to capture the reconfigurations that are allowed within an architectural pattern. They prove that if the initial Bigraph and reaction rules preserve the constraints defined by Σ-sorted Bigraphs then the final Bigraph also does so. Their conformation algorithm essentially checks whether an initial Bigraph and reaction rules can generate a given Bigraphical instance.

The paper is an innovative application of Bigraphs and their reaction rules to express constrains on an architectural pattern. It demonstrates the utility of being able to write one's own reaction rules in Bigraphs to express dynamic reconfigurations in a system.

3) Another paper (Chang et al., 2008a) by the same authors as above expands the above two papers by combining an environment model (where environment is a set whose each element is a vector), and a reconfigurable architecture model (expressed in Bigraphs) into a self-adaptive software model. They propose an algorithm that tests if policies maintain the correctness of the self-evolving software and illustrate their ideas by applying them to a grid application case study. This work combines Bigraphs with automata in an interesting way.

4) The paper (Chang et al., 2008b) by the same research group as above adds to their work by innovatively using the structure of Bigraphs to append context to reaction rules.

5) Wang et al.'s paper (Wang et al., 2010) expresses an Aspect oriented Dynamic Software Architecture (AODSA) with Bigraphs. They have proposed a minor extension of Bigraphs and model components, connectors, aspects, component information managers, connector information managers, and aspect information managers with Bigraphical place graphs. Reaction rules model the dynamic evolution of AODSA. A series of operations such as add aspect, delete aspect, modify aspect and weave aspect are modelled as a single reaction rule expressing the overall dynamic evolution. They have illustrated these ideas by applying them to an aspect oriented client server system.

The inclusion of aspects as part of a Bigraph model is an important original contribution of this paper.

6) The paper by Blackwell (Blackwell, 2011) uses Bigraphs to model system security issues instead of using them to only model mechanisms such as cryptographic protocols. The static structure of the Bigraph models the supervisory control and data acquisition network (SCADA). SCADA is used to manage power plants, sub stations and transmission lines. The system-wide issues modelled by reaction rules include remote logical attacks, transitive attacks, physical attacks, insider attacks, multilevel attacks, consequential impact of attacks, Malware attacks and control attacks.

This application shows the versatility of Bigraph's abstractions- in particular the ease with which discrete physical and logical entities as well as their dynamics can be expressed.

7) The paper by Xue et al. (Xue et al., 2011) models an abstraction slider with CCS encoded with Bigraphs. Abstraction slider (Polyvyanyy et al., 2008) is a mechanism to specify the level of abstraction to express a process model. Xue et al. classify each element of a business process into a slide depending upon its location in the Bigraphical place graph. Different slides can be composed together depending upon the level of abstraction chosen on the abstraction slider.

Although an interesting application of Bigraphs, Xue et al. have not explained the advantages to be had by such an approach.

8) The Calder and Sevegnani paper (Calder and Sevegnani, 2012) investigates runtime verification for event-driven systems. Their example system is the Homework Network Management System (Sventek et al., 2011) which is used to support non-expert users in installing and managing wireless home networking. The Homework system is enhanced by the authors with runtime verification capabilities. The results of the verification are fed back to the users using graphical representation of Bigraphs and if required also to the network. Network topologies are modelled with the structure of Bigraphs. Event, access control policy enforcements and revocations are modelled with Bigraphical reaction rules. Properties of the system are verified using predicates encoded as instances of Bigraph matching. They have used an enhanced version of Bigraphs (Sevegnani and Calder, 2010) that can represent location overlap. Consider Figure 2-28, which shows a Bigraph Model of a wireless local area network (WLAN). M represents a machine linked to the router R. M is also linked to its wireless signal represented by S_M. Similarly, R is linked to its wireless signal S_R. The two wireless signals can overlap and this is captured in the Bigraph.

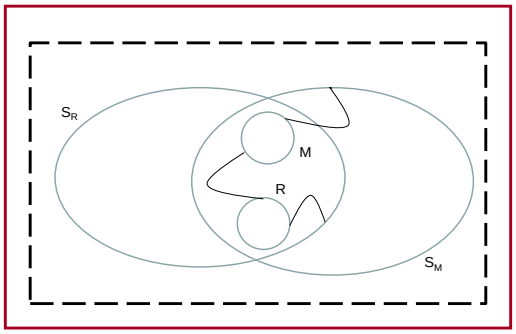

FIGURE 2-28:BIGRAPH MODEL OF A WLAN (CALDER AND SEVEGNANI, 2012).

Apart from our work, this is the only work to the best of our knowledge that deals with runtime issues (though not with models at runtime issues) when modeling with Bigraphs. Other projects

that use Bigraphs as the modeling language do so at the system *design* stage rather than at the system *execution* stage.

2.2.3 OPEN RESEARCH QUESTIONS

None of the papers discussed above have used Bigraphs to *construct* a model at runtime to deal with volatility in service compositions running on a mobile device. Thus, from the papers discussed, the question that emerges is:

How do we use Bigraphs to construct a model at run time?

More specifically, none of the surveyed papers have dealt with the following issues:

a) Instead of using Bigraphs to model systems for simulation, how do we use Bigraphs to express a model that is *causally* connected to a running system?
b) What are the best practices to use Bigraphical abstractions mapped to a programming language to model at runtime a real-world system?
c) Can Bigraphical abstractions be used to implement standard system techniques like caching, delayed-write, pre-fetching? These techniques will be needed to deal with a bottleneck of requests arising out a high rate of reconfigurations in an implemented system.

These are open questions that represent the 'gaps in knowledge' in the surveyed literature.

2.2.4 OUR TAKE-OFF POINT

We have discussed above the papers that use Bigraphs in practical applications. We wish to point out the following approaches from some of these papers that we will also explore in this thesis to see if they are appropriate for our research question:

1) We will use Lars Birkedal et al.'s (Birkedal et al., 2006) approach of the Plato-graphic Model to design our two-layered model. We discuss which of the ideas of Plato-graphic models we use in Chapter 4.

2) We concur with Henson et al. (Henson et al., 2012) that Bigraphs could help us abstract away all the underlying mathematical complexities in a process algebra while still enabling us to utilize the rigor that comes with such approaches when we model real-world scenarios. This of course does not mean that the complexities in themselves can be wished away. Rather Bigraphs present an *interface* that can be used by programmers without going into the *implementation* of

those complexities. Our implementation using MiniML will be a test of this simplicity of Bigraphs.

3) The idea of expressing image schemas with Bigraphs (Walton and Worboys, 2009) represents a systematic approach to modeling an external environment. In this thesis, we explore the runtime phenomenon in the external environment that could be modelled with Bigraphs.

4) Including different views of the same system in one model as discussed by Pereira et al. (Pereira et al., 2012) has become important (Aßmann et al., 2012). Each model represents a specific view of the system. At runtime we might need to manipulate the different views represented by different models of the same system to keep it running properly. We present our techniques to include two views in the same model with Bigraphs in this thesis using the ideas of the Plato-Graphical model (Birkedal et al., 2006).

5) We will examine the appropriateness of using Bigraphs to support a system at runtime as has been done by Calder and Sevegnani (Calder and Sevegnani, 2012) .

2.2.5 SECTION SUMMARY

In this section we have given the relevant background for Bigraphs, their minor extension the Plato-Graphic model, and the BPL Tool. We have then surveyed the literature and shown that Bigraphs have not been used to *construct* a model at runtime. Finally, we have discussed the take-off point for our thesis- in particular the idea of the Plato-Graphic model.

2.3 MODELS AT RUNTIME: A NEW ARCHITECTURE

A model at runtime is defined as *"a causally connected self-representation of the associated system that emphasises the structure, behaviour or goals of a system from a problem space perspective"* (Blair et al., 2009).

"*Abstractions of the problem space express designs in terms of concepts in application domains such as telecom, aerospace, healthcare, insurance and biology*" (Schmidt, 2006).

Abstractions of solution space express designs in terms of computing technologies themselves – in terms of registers and pointers for example in assembly languages and the C programming language.

We firstly explain the concept of models at runtime which is the architecture that we follow in this thesis. Secondly, we discuss the papers that use architectural models at runtime to support dynamic adaptation and software evolution. Thirdly, we discuss the open research questions that

the models-at-runtime community is exploring. Finally, we discuss the take-off point of our thesis with respect to models-at-runtime.

2.3.1 EXPLANATORY BACKGROUND FOR MODELS AT RUNTIME

The concept of a model has often been discussed in software engineering literature. As explained by Waddington and Lardieri (Waddington and Lardieri, 2006), *"rather than replicating abstractions that programming languages provide, models abstract upon "selected" elements of the implemented complex system"*. See Figure 2-29.

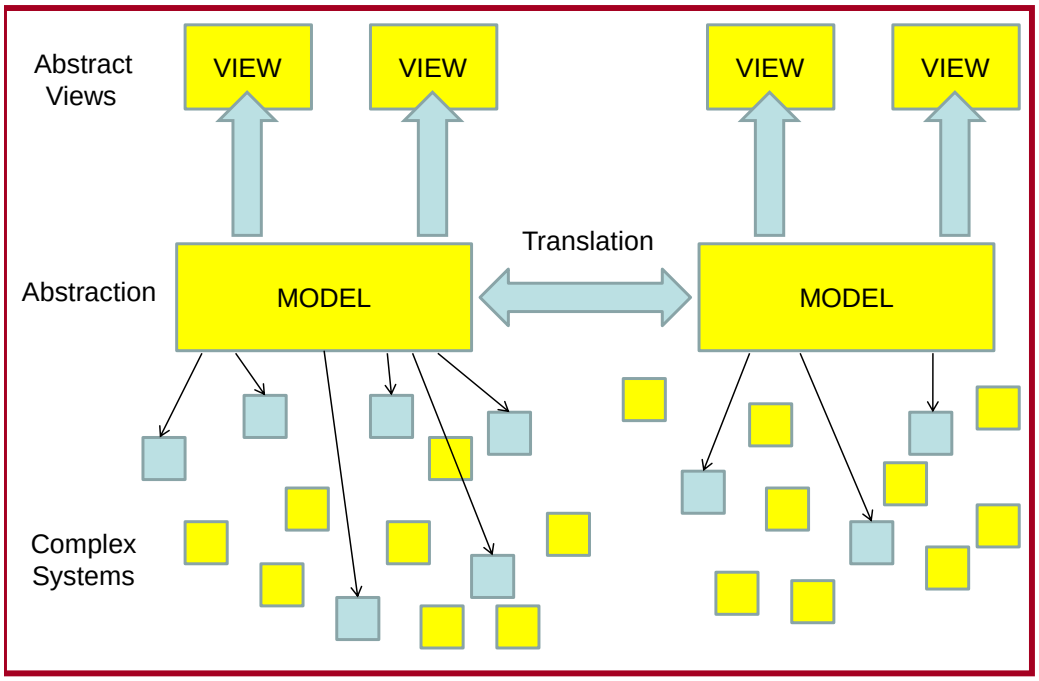

FIGURE 2-29: RELATIONSHIP BETWEEN VIEWS, MODELS AND IMPLEMENTATION.(WADDINGTON AND LARDIERI, 2006).

"*Abstraction is a special case of separation of concerns wherein we separate the concern of important aspects from the concern of the less important details*" (Ghezzi et al., 2002). Models can play two keys roles by applying abstraction (Brambilla et al., 2012):

a) Reduction feature: models capture only selected elements of the implemented complex system.

b) Mapping feature: models can sometimes be based on a system which is considered a prototype of a class of systems. Then the model is said to have generalized the prototype system to a class of many systems.

This concept of a model has been used in developing Model-Driven Engineering (MDE) technologies. MDE technologies have been described by Schmidt (Schmidt, 2006) as *"offering a promising approach to address the inability of third-generation languages to alleviate the complexity of platforms and express domain concepts effectively"*.

MDE techniques manage complexity by having different levels of abstraction and transforming one level of abstraction to another (Morin et al., 2008). These techniques as applied to models at runtime include (Morin et al., 2008):

1) Automatic generation of reconfiguration commands and scripts,

2) Managing the adaptation of a complex system at a more abstract level.

However, models-at-runtime research differs from MDE research in one crucial aspect. Models at runtime research focuses on runtime models in contrast to MDE research that has focussed on design-time models (Bencomo, 2009). These models are abstracted representations of the running system and are causally connected to it (See Figure 2-30).

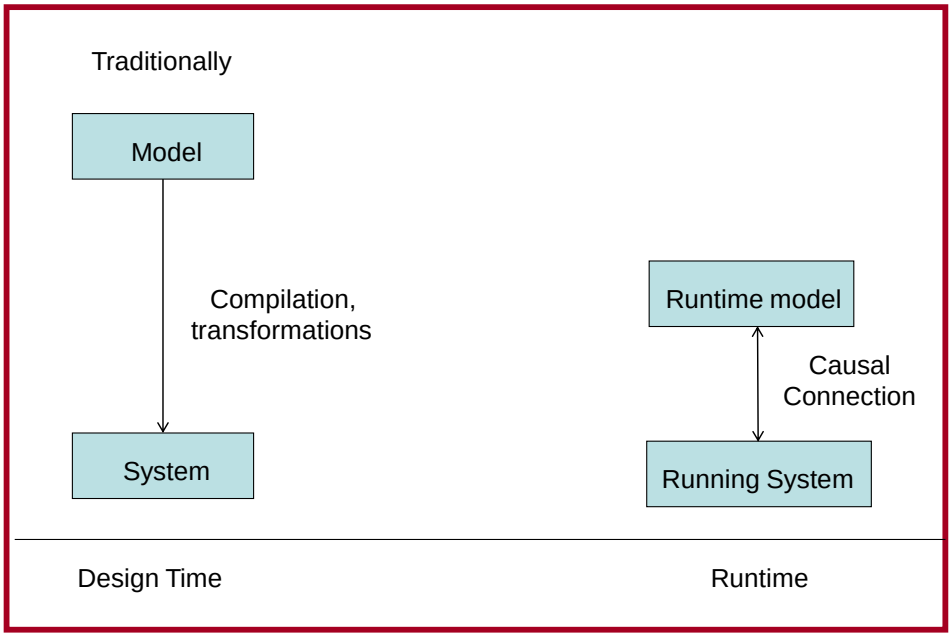

FIGURE 2-30: DESIGN MODELS VERSUS RUNTIME MODELS (BENCOMO, 2009).

This causal connection is essential because models should be able to provide up-to date information to aid in deciding an appropriate adaptation strategy. Moreover, because of the causal connection, adaptations can be effected at the model level. This is similar to models used

by the reflections research community. Borrowing from the research done on reflection in programming languages, the runtime model and the running system could either be one and the same or be different entities (Gjerlufsen et al., 2009) (See Figure 2-31).

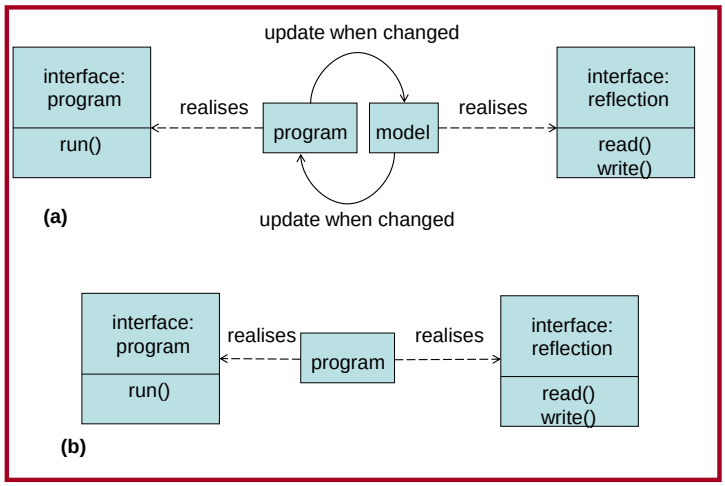

FIGURE 2-31: TWO BASIC APPROACHES FOR RUNTIME SELF-REPRESENTATION THROUGH REFLECTION. (a) A MODEL CAUSALLY CONNECTED TO THE PROGRAM;(b) MODEL AND THE PROGRAM ARE ONE AND THE SAME ENTITY(GJERLUFSEN ET AL., 2009).

However, in contrast to the reflection community's models, models at runtime are based on abstractions of the problem space rather than solution space and hence are at a higher level of abstraction.

As we have seen, models at runtime are essentially abstractions of runtime phenomenon and the various dimensions of models at runtime include (Blair et al., 2009):

1) Structure versus Behaviour: Models could focus on the structure of the system emphasizing how the software is currently constructed: in terms of objects, inheritance relationships and invocation pathways; components and their connections; or aspects and their pattern of weaving. Behaviour models seek to capture how the system executes in terms of flows of events or traces through the system, or the arrival of events and their pathway to execution – arriving, en-queuing, selection, dispatching and so on.

2) Procedural versus Declarative: A procedural model captures the actual structures or behaviours in the system- basically the models capture *how* some aspect of the system works. On the other hand, declarative models seek to capture the system goals- basically the models capture *what* a system does.

3) Functional versus non-functional: Models designed for runtime tend to capture functional properties of a system. Functional properties describe specific functions of the system such as the result of a computation (Cheng et al., 2014). Non-functional properties such as performance and security also need to be captured. Non-functional properties describe the operational qualities of the system such as availability, efficiency, performance, reliability, security etc. (Cheng et al., 2014).

4) Formal versus informal: Models could be based on mathematics of computation or from consideration of programming models or domain abstractions. Formal models support automated reasoning about system's state.

2.3.2 USING ARCHITECTURAL MODELS AT RUNTIME TO SUPPORT DYNAMIC ADAPTATION AND SOFTWARE EVOLUTION

We wish to use architectural approaches to design our model at runtime. Therefore, we now discuss the papers that use architectural models during runtime to support dynamic adaptation and software evolution.

1) Oreizy et al.(Oreizy et al., 1999) present a comprehensive view of an architectural based approach for software adaptation and evolution. In particular, they argue that to support adaptive changes at the architectural level it should be possible to change both their components as well as their interconnections. Moreover, dependencies on environment also need to be explicitly stated with architectural formalisms. They emphasize that the developers of such a system should keep in mind the following issues:

i. What are the conditions under which the system adapts?
ii. Should an open-adaptive or a closed adaptive system be designed? They define a system as being open adaptive *"if new application behaviors and adaptation plans can be introduced during runtime"*. A system is closed adaptive *"if it is self contained and not able to support the addition of new behaviors"*.
iii. How much autonomy must a system have?
iv. A cost-benefit analysis of the adaptive mechanisms should be undertaken.
v. What should be the frequency of adaptation?
vi. How current should the information based on which the adaption decision is taken be?

Bencomo (Bencomo, 2009) has pointed out that if a system is open-adaptive, it is easier to add on a reflective mechanism to it.

2) Dowling and Cahill's paper (Dowling and Cahill, 2001) proposes an architectural meta-model that reifies a system's architecture as a *"typed, directed configuration graph with*

interfaces as vertices, labeled with component instances, and edges as connectors". The transformation of the configuration graph models a reconfiguration of the system architecture. Their approach enables the replacement of components in a CORBA-based system.

3) Garlan and Schmerl (Garlan and Schmerl, 2004) advocate guaranteeing consistency between an architecture model and its implementation at runtime. This can be done by *"monitoring the running system and translating observed events to events that construct and update an architectural model that reflects the actual running system"*. This updated architecture is then compared to the correct architectural model. If inconsistencies are discovered, corrective adaptation is triggered. Thus a constantly updated architectural view that depends on the system's property of interest is maintained. They also argue that mechanisms to effect adaptation must be separate from the system itself. They describe their Rainbow framework (Garlan et al., 2004) which supports the use of an architectural model at runtime. We discuss the Rainbow framework next.

4) Garlan et al.(Garlan et al., 2004) have proposed the Rainbow framework which describes the usage of architectural models for system monitoring and reflection at runtime. They use an external model to monitor and if required modify a system dynamically. The system's monitored events are translated into constructing and updating the architectural model to reflect the actual running system. If an inconsistency is found in a running system, appropriate adaptation commands are issued. They adapt the notion of an architectural style which includes four set of entities: component and connector types, constraints on permitted composition of elements, properties of the component and connector types, and analysis on systems constructed with different architectural styles. They describe two case studies: A web-based client-server system and a video conferencing system.

5) Cazzola et al. (Cazzola et al., 2004) propose a reflective architecture for dynamically evolving an object-oriented system's structural and behavioral aspects. They discuss their event-condition-action rules and a decision engine to control the evolution of the system. A case study of an urban traffic control system is described to show the applicability of their ideas in a real world system.

6) Floch et al. (Floch et al., 2006) describe the project MADAM (Mobility and Adaptation enabling middleware) which uses architectural models at runtime for developing adaptive systems for mobile applications. Generic middleware components are used to reason about and control adaptation decisions. They compare the actual running system with new variants of architectural models that have been derived from utility functions. The utility functions encapsulate their goal policies for runtime adaptation.

7) Caporuscio et al.'s paper (Caporuscio et al., 2007) uses software architecture as the abstraction of the system that they are modeling in a framework for performance management. Their approach is based on monitoring and model based performance evaluation. Runtime monitoring information is used to instantiate architectural models to inform a reconfiguration decision. The new target configuration is generated through a combination of reconfiguration steps that are pre-determined.

8) The paper by Bencomo et al.(Bencomo et al., 2008) describes the Genie Toolkit that can be used to support a reconfigurable component based system's modeling, generation and operation. Two kinds of models are generated by Genie: architectural models and transition state models. The artifacts generated from the models by Genie can be used by reflective middleware to support adaptation at runtime. The artifacts are XML configuration files that can be dynamically inserted into the running system.

9) Sykes et al. (Sykes et al., 2008) organize their architecture into three layers to support adaptation (See Figure 2-32). The actions that require an immediate feedback are performed by components at the lowest level whereas those that require long reflection are performed by mechanisms at the uppermost level. At the lowest level, domain specific software components reside. At the middle layer, mechanisms for plan execution, assembly of components, and replacing components reside. At the uppermost layer mechanisms for converting goals, expressed in temporal logic, into reactive plans reside.

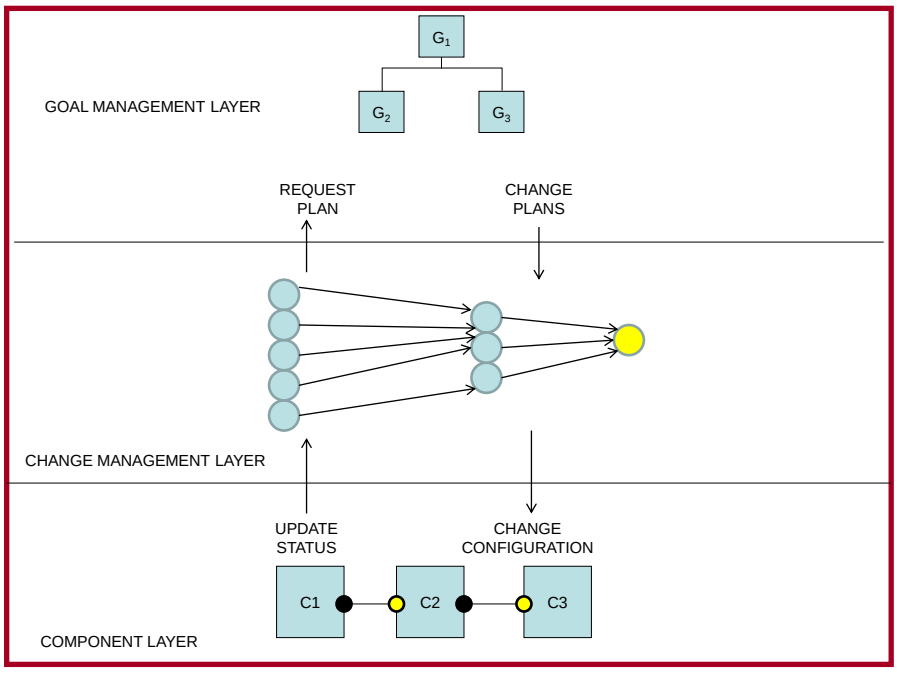

FIGURE 2-32:THREE LAYERED CONCEPTUAL MODEL (SYKES ET AL., 2008).

10) Morin et al. (Morin et al., 2008) have used aspect oriented and model driven techniques to deal with the complexities arising out of the construction and running of adaptive systems. They use aspect oriented techniques to reduce the number of possible system configurations. Model driven techniques are used to generate configuration scripts and to manage the adaptation at a higher level of abstraction than traditional techniques.

We focus on how they have used models at runtime since that part of their work is the inspiration for our architecture.

Morin et al.'s runtime model is causally connected to the running system. The running system can be managed by controlling the model. The model's causal connection is shown in Figure 2-33. The reference model shown in the Figure 2-33 is generated by reflection over the running system. The running system is observed by listeners that update the reference model. The reference model is transformed to the modified model by a reasoning framework responding to changes in context that trigger adaptation requirements. The modified model is then compared to the reference model to produce diff and match models. The diff and match models specify respectively the differences and similarity between models. An analysis of the model results in an instantiation of reconfiguration commands. These commands are used to add or remove bindings and/or components. The commands are ordered by priority and executed by the platform to effect adaptation. To verify that all adaptation commands have been executed successfully, the new reference model (automatically derived through reflection) is checked against the target configuration model.

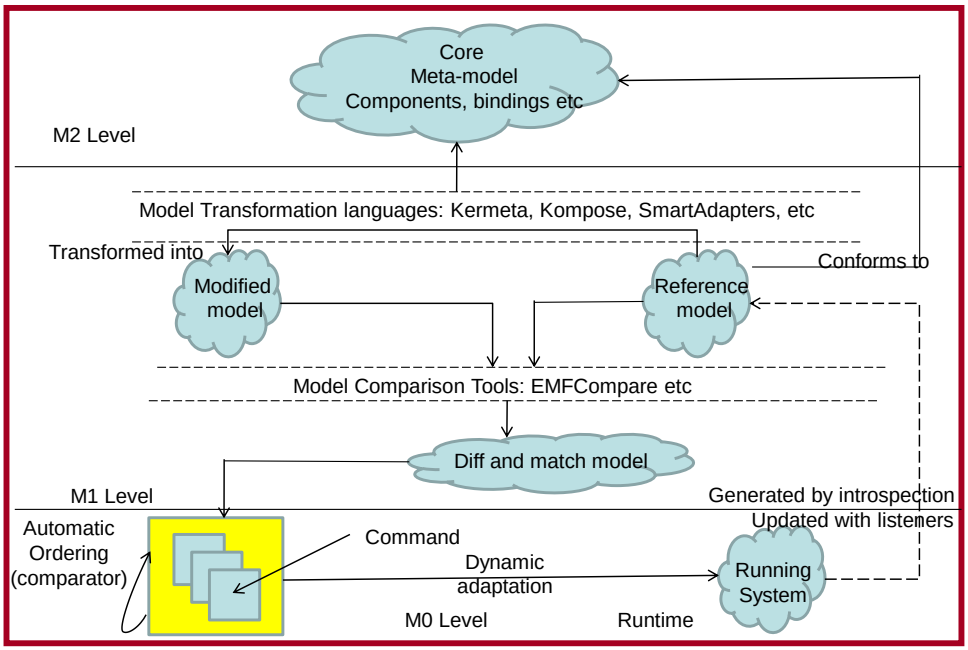

FIGURE 2-33: MORIN ET AL.'S RUNTIME MODEL (MORIN ET AL., 2008).

11) The paper by Georgas et al. (Georgas et al., 2009), represents an adapting system's architectural configurations at runtime with configuration graphs. The goal of this system is to allow human input into the adaptation control loop. This is aided by the configuration graph's monitoring and recording of the necessary information about adaptation.

12) Ramirez and Cheng (Ramirez and Cheng, 2009) propose an approach called Plato-MDE where they use evolutionary computation-based techniques to generate the target system's configuration model at runtime to deal with changes in requirements and environment. Thus, developers need to only specify the relative importance of functional and non-functional concerns. Since Plato-MDE evolves target system models at runtime, the solution space is eventually dominated by better solutions. They demonstrate an application of Plato-MDE to dynamic reconfiguration of an overlay network for diffusing data to a collection of remote data mirrors.

13) Ferry et al. (Ferry et al., 2009) present the use of weaving of aspects of assembly to effect a reactive model transformation at runtime. They define Aspects of Assembly *"as pieces of information describing how an assembly of components will be structurally modified, thus keeping black-box property of components"*. Their example system is a service composition running on a Ubicomp device. They decompose adaptation into three transformations between different models of their approach. They show a way to use meta-models to enforce conformity

of their transformation rules. They have used the Service Lightweight Component Architecture (Hourdin et al., 2008) to dynamically orchestrate and compose services for devices. However, they do not use a formal language to express their model at runtime.

14) The paper by Elkhodary et al. (Elkhodary et al., 2009) proposes a software adaptation framework called Feature-Oriented-Self-Adaptation (FUSION). A system's requirements are broken down into meaningful units of functionality which they call features. A feature is mapped at design time to a part of the underlying software architecture that realises it. They point out the following two ideas that emerge out of their work: : *"(1) features allow representation of the engineer's knowledge about some facets of the system that can be used to enhance the adaptation logic, and (2) features can serve as an abstraction to deal with the heterogeneity of the underlying architectural models, analytical algorithms, and implementation platforms"*.

15) Morin et al. (Morin et al., 2009) use architectural models at runtime to support dynamic software product lines (DSPL). Their goal is to reduce the number of artefacts to support the evolution of adaptive systems by leveraging aspect-oriented and model-driven techniques. They design four aspects of a dynamically adaptive system: its variability, the system's environment and context, adaptation logic and the system architecture. Aspect-Oriented techniques are used by them to automatically construct architectures by composing aspects that model features. Model- Driven techniques are used by them to produce adaptation commands.

2.3.3 OPEN RESEARCH QUESTIONS FOR MODELS AT RUNTIME

The models at runtime research community organized the Models@run.time seminar at Dagstuhl in November, 2011 to discuss the open research questions for future work (Aßmann et al., 2012). From those questions, this thesis addresses the following by using Bigraphs as a language for *constructing* a model at runtime:

a) How can a runtime model provide a means to store and retrieve information about the environment and the system?
b) To facilitate adaptation, how can a model at runtime provide a representation of the current state and reconfiguration rules?
c) How can a runtime model enable us to reason about operating environment and runtime behaviour to determine an appropriate form of adaptation?
d) How can a runtime model provide meta-information along the following two dimensions: a) efficient use of time, b) location dependency?

e) How to combine two models of runtime into one? This necessity of combining multiple models *"arises because of the need to manage multiple concerns, for example, performance, reliability, and functional concerns. Each concern typically requires specific models that are able to capture the individual concern and to provide a basis for reasoning about it*" (Bennaceur et al., 2014).

f) How to select one appropriate adaptation command out of many such commands at runtime?

2.3.4 OUR TAKE-OFF POINT

The papers discussed above in section 2.3.2 have supported dynamic adaptation and software evolution by using architectural models at runtime. Some of the design ideas that we will explore in this thesis to construct our system are:

1) A layered approach to designing a runtime model is required to tame the complexity of runtime phenomenon (Sykes et al., 2008).

2) Running systems should be open-adaptive (Oreizy et al., 1999) to make it easier for reflective systems to be added.

3) A runtime model that expresses the architecture of the running system should be causally connected to it (Blair et al., 2009, Morin et al., 2008, Morin et al., 2009).

4) System events need to be monitored. These monitored events need to be translated into events that construct and update a model that reflects the actual running system (Garlan and Schmerl, 2004, Garlan et al., 2004, Morin et al., 2008).

5) The model needs to be transformed into a target model because the system that the original model represents has changed. The changed system needs to be represented by the target model. The differences between the target model and the currently running system should be translated into adaptation commands (Morin et al., 2008, Morin et al., 2009).

2.3.5 SECTION SUMMARY

In this section, we have explained the ideas behind models-at-runtime. We have surveyed the literature that supports dynamic adaptation and software evolution with architectural models at runtime because we want to use such models at runtime to organize the design of our system. We have discussed the relevant research questions that came out of discussions held in the Dagstuhl seminar by the models at runtime research community(Aßmann et al., 2012). Finally,

we have discussed the take-off point for our work in particular the ideas in the paper by Morin et al.(Morin et al., 2008).

2.4 VOLATILITY: THE EXAMPLE PROPERTY

Volatility is the example property that our system needs to tackle. From a distributed systems perspective, volatile systems include mobile and hand-held computing systems, ubiquitous computing systems, wearable computing systems, context-aware computing systems, tangible computing systems and augmented reality systems (Coulouris, 2012).Volatility is manifested in a system in the following ways:

1) Device and communication link failures,

2) Variation in the properties of communication such as bandwidth,

3) Creation and destruction of associations which are logical communication relationships between software components resident on the devices.

As Coulouris emphasises (Coulouris, 2012), volatility is not the defining property of the systems mentioned above since other systems such as a file-sharing peer-to-peer system also show forms of volatility. However, in contrast to other systems, volatile systems show all of the above mentioned forms of volatility at once. Moreover, the rate of change in a volatile system is much higher than in a distributed system.

Bardram and Friday (Bardram and Friday, 2010) have pointed out that besides the above, volatility in Ubicomp systems is also caused by changes in topology, routing and host naming. Caceres and Friday (Caceres and Friday, 2012) describe volatility as resulting from a changing *"set of users, devices and software components in an environment-far more frequently in ubicomp systems than in conventional distributed systems"*.

Thus one of the key things about volatile systems is the high rate of change that is occurring. However, to the best of our knowledge, the literature on volatility does not specify *how* high this rate of change is. At best, it is described as being "at least one such change occurring at any one time" as in the following (Coulouris, 2012):

"*An important difference that may arise between volatile systems is the rate of change. Algorithms that have to cope with a handful of appearing or disappearing of components a day (e.g., in a smart home) may be very differently designed from those for which there is at least one such change occurring at any one time (e.g., a system implemented using Bluetooth communication between mobile phones in a crowded city)*."

2.5 UBIQUITOUS COMPUTING SYSTEM SERVICE COMPOSITION FAULTS: THE EXAMPLE SYSTEM PROBLEM

Our example system is a service composition running on a mobile device. We now discuss the faults that such a service composition could suffer from. Our Bigrapical model at runtime will attempt to provide a way to recover from the service composition faults in a composition running on a mobile device. Thus ubiquitous computing system service composition faults is the example system problem we wish to tackle with our proposal for a Bigraphical model at runtime.

Service is defined as *" a platform independent, loosely coupled, self contained, programmable application that can be described, published, discovered, coordinated and configured for the purpose of developing distributed interoperable applications*" (Papazoglou et al., 2007). A similar definition is given by Sumi Helal (Helal, 2010): *"Services are software components with well-defined interfaces and they are independent of the programming language and the computing platforms on which they run"*. Thus, Service-Oriented-Computing (SOC) is independent of specific technologies such as Web Services or Event-Driven systems (Poslad, 2009) and services are components with well defined interfaces (Helal, 2010),(Alonso et al., 2010). Service composition refers "*to the development of customized services by discovering, integrating and executing existing services"*(Chakraborty et al., 2005).

Bronsted et al.(Bronsted et al., 2010) have identified managing contingencies as one of the main goals of service composition for ubiquitous computing. In a ubiquitous computing environment often called smart spaces (Coulouris, 2012), devices and services running on them might suffer from faults because of device or battery failure resulting in their unpredictable availability. Moreover, the faults might sometimes be induced because of device mobility.

K.S. May Chan and co-workers have proposed a fault taxonomy for web service composition (Chan et al., 2007b). Although our example system is a service composition running on a mobile device, Chan et al.'s taxonomy is relevant to us. This is because the same faults as those of web service composition will also be manifested in a service composition running on a mobile device albeit at a higher rate (Coulouris, 2012).

Chan et al.'s taxonomy for web services adapts Avizienis's broader taxonomy of faults for any computing system (Avizienis et al., 2004). Faults cause failure in a system and can be categorised into:

1) Physical Faults.

2) Development Faults.

3) Interaction Faults.

We now discuss each of these categories:

1) Physical Faults: These are caused by a network or server side failure. Communication infrastructure exceptions and incorrect operation of the hosting server's middleware are also categorized as physical faults. An *unavailability* fault occurs when one of the services of the composition becomes unavailable.

2) Development Faults: These can occur because of human developers, development tools or production facilities. The faults under this category include:

i. *Interface change* fault: This fault occurs if one of the services in the composition updates its interface without warning.
ii. *Workflow inconsistency* fault: This fault occurs when the workflow description and interface of a service do not match.
iii. *Parameter incompatibility* fault: This fault occurs if a service is provided with either incorrect arguments or incorrect parameter types as input.
iv. *Non-deterministic action* fault: This occurs when a service has a non-deterministic outcome or return value.

3) Interaction faults: These can occur and propagate between services during the execution of the composite service. This category is further sub-divided into the following sub-categories:

i. *Content Faults*: These occur when the content that is delivered by a service is different from the service description. Content faults are of the following types:
   a) Incorrect service fault: This occurs when a service provider delivers a service which has not been requested.
   b) Misunderstood behavior fault: A service requestor misunderstands a service description and requests for a service that is not being provided by the service provider.
   c) Response error fault: This occurs when a service is provided with a correct input but responds with incorrect results.
   d) Service-Level-Agreement (SLA) fault: This occurs when a service provided by a service provider does not comply with the SLA.
   e) QoS fault: This fault occurs when a service provided by a service provider is not of good quality in terms of speed and information.

ii. *Timing Faults*: These are the second type of interaction faults and occur when a service's time of arrival or timing of delivery results in the composition not complying with the originally specified functional requirement. Timing faults are of the following types:

a) Incorrect order fault: This is caused by a slow network and can result in message packets arriving in an order different to the order in which they were sent.
b) Timeout fault: This is again caused by a slow network when a service waiting for a message packet 'times out'.
c) Misbehaving workflow fault: This is caused if the workflow of a composite service is not correct or an individual service used by a composite service is incorrect or one of the services forming part of the composition does not work well with other participating services.

Chan et al. combine the above faults and their observed effects into a fault taxonomy as shown in the Figure 2-34. The three categories of physical, development and interaction faults are shown along the top of the taxonomy and divided into the sub-categories that we discussed above. The bottom of the taxonomy shows the observed effects. An observed effect can be triggered in more than one way as shown in the taxonomy.

Chan et al. also utilize Avizienis's sixteen elementary fault classes and identify the following six that are relevant for web services:

1) Phase of occurrence viewpoint:

i. Development faults occurring during system development or maintenance.
ii. Operational faults occurring during service delivery.

2) System boundary viewpoint:

i. Internal faults that originate inside the system boundary
ii. External faults originating outside the system boundary. These faults are then propagated into the system because of interaction or interference.

3) Fundamental systems viewpoint:

i. Hardware faults: These originate in the hardware or effect the hardware.
ii. Software faults that affect programs or data.

The six faults mentioned above are plotted along the vertical axis on the left of the taxonomy as shown in the Figure 2-34.

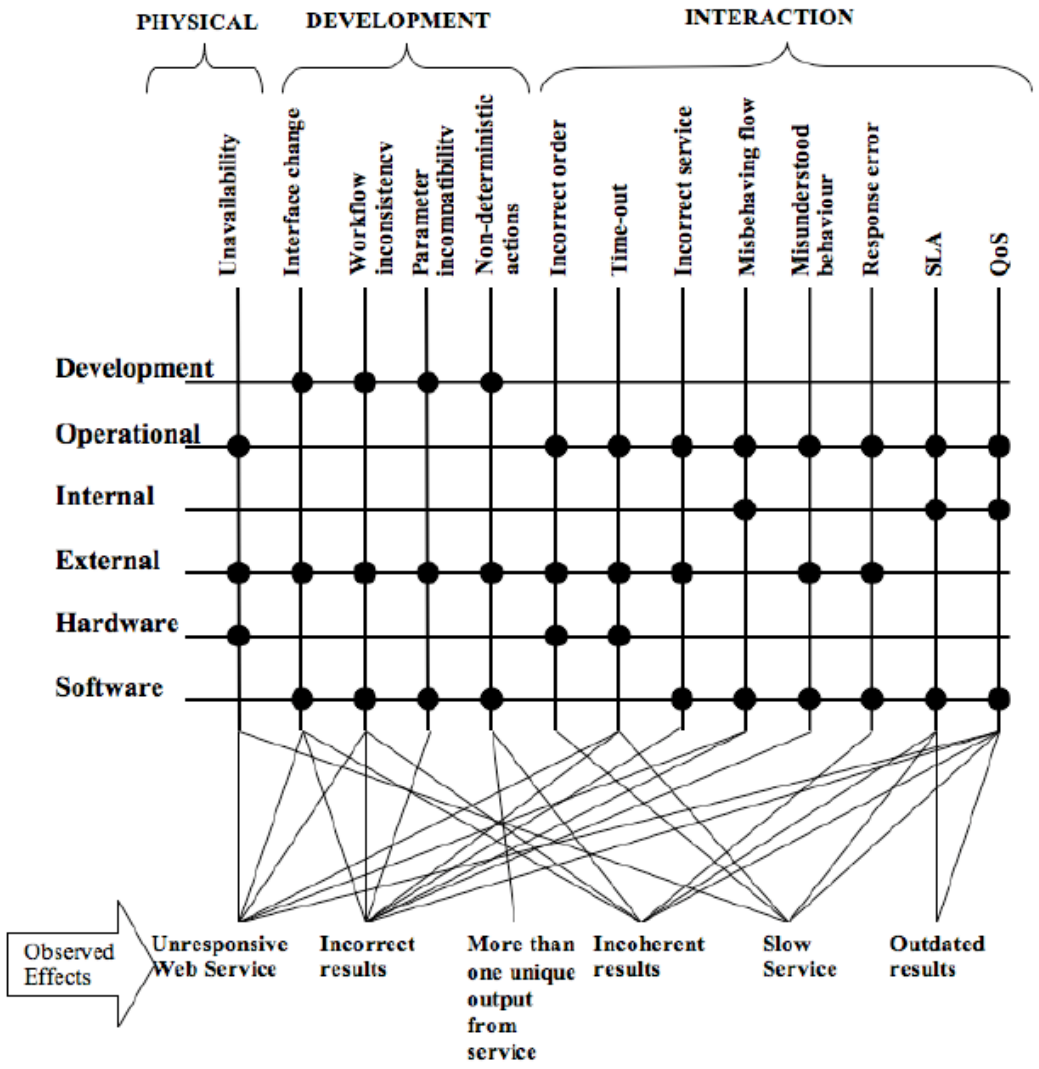

FIGURE 2-34: TAXONOMY OF FAULTS COMBINED WITH OBSERVED EFFECTS (CHAN ET AL., 2007B).

Chan et al.'s goal is to analyze the observed effects so as to narrow down the possible faults that could have caused a failure in a web service composition.

In conclusion in this section, we have discussed how faults occurring in a service composition are categorized and how the observed effects can be mapped to the possible faults as discussed by Chan et.al. To the best of our knowledge, this fault taxonomy is the most appropriate classification for the service faults occurring in a service composition running on a mobile device.

2.6 CONCLUSIONS

In this chapter, we have discussed the literature relevant to our research goal of exploring the appropriateness of Bigraph's abstractions to *construct* a model at runtime to tackle the problem of volatile service composition running on a mobile device.

We have described Bigraph's abstractions including placing, linking, interfaces, node classifications and signatures. Additionally, we have shown how larger Bigraphs could be constructed from smaller ones using the three operations of MiniML which is a subset of SML. These operations are composition, parallel product and prime product. We have also discussed how to represent a dynamic reconfiguration of Bigraphical structure with reaction rules. Next, we have discussed the Plato-graphical Model, which is a minor extension of Bigraph. We have then explained the architecture of BPL Tool, which we have used to implement our system. We have organized the literature that discusses using Bigraphs to represent architecture of software systems into not necessarily mutually exclusive categories.

From this literature, we have identified the gaps in knowledge that we propose to address in this thesis.

Furthermore, we have discussed those ideas from the literature that are the take-off point for our thesis.

We have then described models at runtime as *"a causally connected self-representation of the associated system that emphasises the structure, behaviour or goals of a system from a problem space perspective"* (Blair et al., 2009).

We have described the papers that use architectural models to support dynamic adaptation and software evolution. From these papers, we identified those open research questions for models at runtime which are relevant to this thesis.

Moreover from the papers surveyed, we have picked out the ideas that are the take-off point for our thesis.

We have also described volatility as a property of a ubiquitous computing system and have pointed out that the effects of volatility occur at a higher rate in ubiquitous computing system.

Finally, we have described how faults occurring in a service composition are categorized into physical, development and interaction faults and how the observed effects can be mapped to possible faults.

3 THE RESEARCH QUESTION AND ITS DESIGN IMPLICATIONS

3.1 INTRODUCTION

In Chapter 2, we critically analyzed the literature and identified those open questions that constitute our take off point for this thesis.

From those take off points, we now present the research question that has emerged and that we will tackle in this thesis. We also discuss the generic requirements for the design of our system and the associated design space.

This chapter is organized as follows: In Section 3.2, we define our research question, discuss the research work done by others that we have used as our starting point, and show why it is worthwhile to answer our research question. Next in Section 3.3, we describe the requirements for our design, which must support strategies to deal with volatile service composition running on a mobile device. Finally, in Section 3.4, we outline the design space for our system by describing the set of decisions that we have taken.

3.2 DEFINING THE RESEARCH QUESTION

We now discuss our research question, how we have leveraged other research work for our design, and show why it is worthwhile to answer our research question.

Our thesis answers the following question:

Are the language abstractions provided by Bigraphs sufficient and appropriate to construct a model at runtime to tackle the problem of volatility in a service composition running on a mobile device?

This research question is a synthesis of the following two issues that have emerged out of our literature review discussed in Chapter 2:

1) *How do we use Bigraphs to construct a model at runtime?*

As discussed in Chapter 2, section 2.2.3, this entails exploring the following questions which we repeat for completeness:

a) Instead of using Bigraphs to model systems for simulation, how do we use Bigraphs to express a model that is *causally* connected to a running system?

b) What are the best practices to use Bigraphical abstractions mapped to a programming language to model at runtime a real-world system?
c) Can Bigraphical abstractions be used to implement standard system techniques like caching, delayed-write, pre-fetching? These techniques will be needed to deal with a bottleneck of requests arising out a high rate of reconfigurations in an implemented system.

2) *Do Bigraphs offer the appropriate language abstractions to address the open research questions being explored by the models at runtime community?*

As discussed in Chapter 2, section 2.3.3, this entails exploring the following questions which we repeat for completeness:

a) How can a runtime model provide a means to store and retrieve information about the environment and the system?
b) To facilitate adaptation, how can a model at runtime provide a representation of the current state and reconfiguration rules?
c) How can a runtime model enable us to reason about operating environment and runtime behaviour to determine an appropriate form of adaptation?
d) How can a runtime model provide meta-information along the following two dimensions: i) efficient use of time, ii) location dependency?
e) How to combine two models of runtime into one?
f) How to select one appropriate adaptation command out of many such commands at runtime?

We have discussed in Chapter 2 that a model at runtime is defined as *"a causally connected self-representation of the associated system that emphasises the structure, behaviour or goals of a system from a problem space perspective"* (Blair et al., 2009). Recall also from Chapter 2 that ubiquitous systems are often characterized as being volatile (Coulouris, 2012), (Bardram and Friday, 2010),(Caceres and Friday, 2012). This includes all of the following properties:

1) Device and communication link failures,

2) Variation in the properties of communication such as bandwidth,

3) Creation and destruction of associations which are logical communication relationships between software components resident on the devices.

On one hand, the need for experimental application of Bigraphs has been pointed out by Robin Milner and Lars Birkedal (Milner, 2009),(Birkedal et al., 2006) among others. On the other hand, the model at runtime community has been exploring appropriate abstractions to deal with complexity arising out of runtime phenomenon (Blair et al., 2009).

As discussed in the previous chapter, the volatile system that will be supported in its adaptation by our proposed Bigraphical model at runtime is a service composition running on a mobile device.

Our thesis is, to the best of our knowledge, the first to identify ways to exploit Bigraph abstractions for expressing a model at runtime.

3.2.1 CAVEATS ON THE SCOPE OF THE RESEARCH QUESTION

We wish to point out two caveats on the scope our research question:

i. Recall from Chapter 2 that as explained by Waddington and Lardieri (Waddington and Lardieri, 2006), *"rather than replicating abstractions that programming languages provide, models abstract upon "selected" elements of the implemented complex system"*. Thus, we do not need to capture *all* programming language abstractions to implement our Bigraph model to succeed in answering our research question. Rather, we need to make a choice of *some* elements of the service composition that we would like to model at runtime.

ii. Elsborg has already pointed out that Bigraphs lack control structures (Elsborg, 2009). His work proposes MiniML, which is syntactic sugar for creating BGVals in SML. One of the reasons given by Elsborg for designing MiniML in this fashion was to gain access to SML's control constructs. Thus, in constructing our Bigraphical model at runtime with MiniML to answer our research question, we acknowledge that we will be accessing the control structures of SML. This will also include SML abstractions that support the control flow such as recursion, function and module. Indeed, the BPL Tool that we will use is itself organised around SML's modules and functions. Also, reaction rules will *not be used to program the model* but to represent transitions in the internal structure of the system and the external environment.

3.2.2 EVALUATION CRITERIA TO TEST IF OUR RESEARCH QUESTION HAS BEEN ANSWERED

To define the scope of our thesis, our evaluation criteria for testing if our research question has been answered are along the following two dimensions subject to the caveats discussed above:

i. Have we been able to *construct* a model at run time that is expressed using Bigraphical abstractions? Such a system will then serve as a proof-of-concept that it is indeed

possible to undertake such a construction. This of course will be a constructive proof of existence.

ii. Can such a Bigraphical model at runtime be in-sync with the real world in terms of the time it takes to respond to the events that are being generated in the real world? Or if they are not in-syc, why not? One of the ways we could do this is by building a test-rig, which will load our Bigraphical model at run time with appropriate events and measure its response times.

3.2.3 THE TAKE-OFF POINT

Our work builds upon work by Lars Birkedal et al (Birkedal et al., 2006) and Morin et al (Morin et al., 2008). We now discuss each in turn pointing out in particular the enhancements that we will implement in our system to answer our research question.

3.2.3.1 ENHANCEMENTS PROPOSED BY US TO THE WORK BY LARS BIRKEDAL ET AL.(BIRKEDAL ET AL., 2006)

We propose to built on Plato-Graphic model (PGM) proposed by Lars Birkedal et al (Birkedal et al., 2006).

The PGM is used by Elsborg to *simulate* (Elsborg, 2009) the Lancaster University's tour guide 'GUIDE' (Cheverst et al., 2000). In contrast, we propose to use PGM-like Bigraphical Reactive systems to construct a *model at runtime* that supports the running of a volatile service composition on a mobile device without user's intervention.

Furthermore, Elsborg (Elsborg, 2009) uses Bigraphical reaction rules to implement queries whereas we propose to use reaction rules to model events that change the structure of Bigraph or to extract information out of it.

Finally as defined by Birkedal et al., PGMs have three layers representing 'World', a 'Proxy' that observes it and an 'Agent' that models the application whereas our proposed model has two layers only. So strictly speaking, our model cannot be called a PGM.

As discussed in Chapter 2, in our proposed approach, similar to Henson et al.'s (Henson et al., 2012) we abstract away all the underlying complexity of Bigraphs. Moreover, we propose to follow Walton and Worboy's systematic approach (Walton and Worboys, 2009) when we model runtime phenomenon. Also, like Pereira et al. (Pereira et al., 2012) we propose to include two views of the system in the same model. Above all, as in the work of Calder and Sevegnani (Calder and Sevegnani, 2012), we propose to use Bigraphs to support a system at runtime.

3.2.3.2 ENHANCEMENTS PROPOSED BY US TO THE WORK BY MORIN ET AL.(MORIN ET AL., 2008)

The architecture that we propose to use is inspired by Morin et al.'s work (Morin et al., 2008) which encompasses ideas in the work done by Garlan and co-workers (Garlan and Schmerl, 2004, Garlan et al., 2004). They specify all the possible variants of a system at design time. These variants are the possible states a system could be in. In contrast, we propose to use reaction rules of Bigraphs to *generate* variants of a service composition at runtime. This affords us greater flexibility in expressing a model at runtime for an infinite number of variants.

Moreover, Morin et al use models at runtime to deal with changes in context whereas we propose to deal with faults occurring in a service composition running on a mobile device.

As discussed in Chapter 2, Sykes et al.'s (Sykes et al., 2008) idea of a layered approach is proposed to be adopted by us to design our runtime model. We assume that the service composition that we have modelled is open-adaptive (Oreizy et al., 1999).

We will give detailed description of the features of our implementation in the following chapters.

3.2.4 APPLICATIONS OF BIGRAPHS AND MODELS AT RUNTIME

We now discus why it is worthwhile to answer our research question in terms of the future role of Bigraphs and models at runtime. Firstly, we explore how Bigraphs are envisaged as a step towards tackling the complexity of ubiquitous systems. Then, we discuss how models at runtime are envisaged as a set of techniques to tackle complexity of runtime systems.

3.2.4.1 FUTURE ROLE OF BIGRAPHS AS ENVISAGED BY MILNER (MILNER, 2006B)

Ubiquitous computing systems consist of a large number of autonomous agents (Milner, 2006b) which could be software based or devices such as microcontrollers with sensors and/or actuators. These agents interact with each other in unpredictable ways using higher level concepts such as trust and move around both physically and logically (for example, a software component binding itself to a new device) in a smart space (Coulouris, 2012). The agents might have knowledge about their environments and be able to negotiate with each other. The agents might also have the ability to be adaptive to the environment.

To be able to tame this conceptual complexity, Milner has proposed a 'tower of models' (Milner, 2006b). The higher level models in this tower express concepts such as trust between

agents. The lower level models implement concepts such as trust by for example having an agent accept data only from a 'trustworthy' agent.

Milner has envisaged Bigraphs as the lowest level foundational model in this tower of models and calls it the Ubiquitous Abstract Machine (UAM) (Milner, 2008b). Bigraphs model the concepts of structure, motion, connectivity and stochastics at this level.

3.2.4.2 FUTURE ROLE OF MODELS AT RUNTIME AS ENVISAGED BY THE MODELS AT RUNTIME RESEARCH COMMUNITY

The models-at-runtime research community envisages using Model-Driven-Engineering techniques to develop models that are abstractions of runtime phenomenon (Blair et al., 2009). Such models could be used to support reasoning, dynamic state monitoring and control of systems at runtime. A user of a system could use models at runtime to understand the behaviour of system at runtime. Moreover, a large variety of software elements could be integrated semantically with the support of a model at runtime. It could also assist in the automated generation of implementation entities which could then be inserted into the system by a user or by the system itself.

In the long term, models at runtime could be used to rectify errors during design. New design decisions could also be implemented as the system is running. Finally, runtime models could be used to aid adaptation decisions and provide meta-information to assist in autonomic decision making.

Through implementing a model at runtime expressed in Bigraphs, we examine if it is appropriate to consider Bigraphs as Ubiquitous Abstract Machines – a foundational model for ubiquitous computing systems.

3.2.5 SECTION SUMMARY

In this section, we have defined our research question as investigating the appropriateness of Bigraphs to construct a model at runtime to deal with a volatile service composition running on a mobile device. We have pointed out the evaluation criteria that we will use to test if we have tackled the research question appropriately. Our starting point is the Plato-graphic model (PGM) which is a minor extension of Bigraphs proposed by Lars Birkedal et al. (Birkedal et al., 2006). We use PGM-like Bigraphs to express a model at runtime. The architecture to support our model at runtime has been inspired by Morin et al.'s work (Morin et al., 2008). We will use this architecture to facilitate the usage of Bigraph's reaction rules to generate variants of service composition structure resulting out of faults triggered by volatility. Finally, we have discussed why it is worthwhile to answer our research question both for the Bigraphs and the models at

runtime research communities. We have shown how Bigraphs can be used as a foundational model in a tower of models to tackle the complexity of ubiquitous systems. We have also shown how a model at runtime can be used as an abstraction of runtime phenomenon to tackle the complexity of runtime systems.

3.3 REQUIREMENTS FOR DESIGN

In this section, we discuss how volatility imposes certain requirements on the design of a model at runtime. We discuss how a high rate of events is a volatile system property that affects the running of a ubiquitous system -in particular a service composition running on a mobile device. As a result of volatility, we need to explore the reconfiguration cycle that such an architecture must support.

As discussed in the scenario in Chapter 1, as the user Alice strolls in a shopping mall looking to buy a pair of jeans, a (volatile) service composition might be running on her mobile device to help her in her shopping.

The volatile system property introduces complexity that any design of a runtime system needs to address. Some of the complexity that needs to be dealt with includes: services malfunctioning at a higher rate due to volatility and a large number of equivalent services being available.

Managing complexity, due to such volatility, through the use of models at runtime has now become an important technique (Aßmann et al., 2012), (Blair et al., 2009), (France and Rumpe, 2007) .

3.3.1 VOLATILE SYSTEMS: AN OPERATIONAL POINT OF VIEW

From an operational point of view (Coulouris, 2012), a volatile system (A system that displays volatility properties- see Section 2.4) exists in a smart space. The smart space is populated with devices that offer services but have limited computing resources and energy supply. These devices suffer from frequent disconnections owing to their limited operating distance or radio occlusions. The devices host software components which frequently change their logical relationships with other components. This frequent change is largely physically driven in volatile systems and results in a high rate of volatile events such as faults.

3.3.2 RECONFIGURATION CYCLE THAT NEEDS TO BE SUPPORTED BY THE ARCHITECTURE

The Bigraph model at runtime of the service composition should support Manel Fredj and co-workers' reconfiguration cycle for composite services that run in a ubiquitous computing environment (Fredj et al., 2006). Our architecture of the model at runtime needs to support this cycle as follows: In Figure 3-1, each rectangular box corresponds to a particular phase in the cycle. Four phases constitute the cycle. In the first phase, the composite service is running properly on the mobile device. We assume that a module outside our system boundary is monitoring the service execution. Next in phase two, a fault is detected in a service forming part of the composition resulting in a need to replace the malfunctioning service. Then in phase three, we choose an appropriate replacement service. Finally, in phase four, we issue the appropriate adaptation commands.

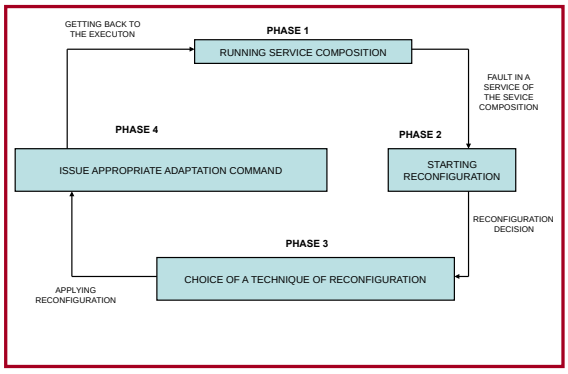

FIGURE 3-1: RECONFIGURATION CYCLE(FREDJ ET AL., 2006).

3.3.3 SECTION SUMMARY

We have discussed in this section that from an operational point of view, volatile systems exist in smart spaces populated by resource-constrained devices that suffer from frequent physically driven disconnection. We have also discussed the four-phase reconfiguration cycle that our architecture will need to support.

3.4 THE DESIGN SPACE FOR TACKLING VOLATILE SERVICE COMPOSITION

Lemos et al. (Lemos et al., 2012) define the design space of a system as *"the set of decisions, together with the possible choices the developer must make"*. We now discuss both our choice

of models at runtime based architecture as well as Bigraphs as a language to construct it that arise from our requirement of designing a model at runtime for a volatile service composition.

3.4.1 OUR CHOICE OF MODELS AT RUNTIME BASED ARCHITECTURE

Replacing a malfunctioning service from a service composition is a form of compositional adaptation. According to Philip McKinley et al (McKinley et al., 2004), *"compositional adaptation enables software to modify its structure and behaviour dynamically in response to changes in its execution environment".* In contrast, parameter adaptation *"modifies program variables that determine behaviour"*(McKinley et al., 2004).

To support compositional adaptation for service compositions at runtime, we need techniques to deal with complexity arising out of a high rate and ill-structured order of occurrence of faults that a service can suffer from and the infinite number of possible reconfigurations of the composition.

As discussed earlier, to deal with this runtime complexity, adaptation mechanisms that leverage software models are being explored by the models at runtime research community (Blair et al., 2009). We have shown in Chapter 2 that models abstract only certain relevant elements of a running system instead of using programming abstractions (Waddington and Lardieri, 2006).

In the process of answering our research question (section 3.2), we seek to also address some of the research goals of the models at runtime community (Aßmann et al., 2012) as discussed in Section 2.3.3.

3.4.2 OUR CHOICE OF BIGRAPHS TO CONSTRUCT A MODELS AT RUNTIME BASED ARCHITECTURE

The models-at-runtime community draws on and extends the lessons learned by the broader Model-driven engineering (MDE) community. We now give a brief overview of those concepts from MDE that we have used in this thesis. Next we situate Bigraphs with this framework of MDE.

3.4.2.1 OVERVIEW OF THE MODEL-DRIVEN ENGINEERING FRAMEWORK

One particular type of design space is the *modelling space*. According to Gasevic et al. (Gasevic, 2006), *"A modelling space defines a conceptual framework to provide an easier understanding of approaches to modelling such as ontologies and the Object Management Group's (OMG) Meta-Object-Facility defined modelling languages such as UML and ODM (Ontology Definition Meta-model)"*.

Modelling spaces are of two types (Gasevic, 2006) : A conceptual modelling space and a concrete modelling space.

According to Gasevic et al. (Gasevic, 2006):*"Conceptual modelling spaces are focussed on conceptual (abstract or semantic) things such as models, ontologies and mathematical logics"*. These spaces correspond to for example the semantics of programming languages.

On the other hand, Gasevic et. al. (Gasevic, 2006) describe concrete modelling spaces as *"materializing (or serializing) concepts of the conceptual modelling space"*. These spaces correspond to for example the syntax of programming languages.

Both the conceptual and concrete modelling spaces can each be represented as a stack of the modelling layers of MDE. Each layer is at a different level of abstraction with the lowest layer being at the lowest level of abstraction. In MDE, the lowest layer M0 represents the system that is being modelled. Above M0 is layer M1 that represents the model of the system. The next layer M2 represents the meta-model and layer M3 represents the meta-meta model. According to OMG's definition (OMG, 2011) , "*A meta-model is a model that defines the language for expressing a model"*.

In an orthogonal arrangement of modelling spaces, *"one modelling space models concepts from another modelling space, taking them as real world things" (Gasevic, 2006).*

We now use these terms to situate Bigraphs within the context of the model driven engineering framework.

3.4.2.2 SITUATING BIGRAPHS WITHIN THE CONTEXT OF MODEL-DRIVEN ENGINEERING FRAMEWORK

As discussed in Chapter 2, we have implemented our system using the BPL Tool (ITU, 2011) which provides a set of SML-constructs called MiniML (ITU, 2007a) which is a subset of Standard ML and can be translated into terms representing Bigraphs (called BGVals). MiniML gives us access to SML's control constructs, which are lacking in Bigraphs. Indeed, one of the stated goals of developing MiniML was to provide such an access to control constructs (Elsborg, 2009) . In Figure 3-2, we show that Bigraphs belong to the conceptual modelling space (Gašević et al., 2009) and represent semantics of the model. The concrete modelling space of Extended Backus-Naur form (EBNF) (which is used to define Standard ML grammar of which MiniML is a subset) is used to implement the conceptual modelling space and represents the syntax in which the model is expressed.

Layers M0, M1, M2, and M3 represent respectively the system being modelled, the model, the meta-model and the meta-meta model as defined in the Model Driven Architecture (MDA).

For Conceptual Modelling spaces, we will be modelling the service composition's architecture (SCA layer) and its environment (WORLD layer). We define both the SCA and WORLD layers in the next chapter.

For the concrete modelling space, the system being expressed in the M0 layer is BgVal which is a low level term language for Bigraphs checked for well-formedness with interface data closely based on elementary Bigraphs and combinators (ITU, 2007a).

Notice the arrangement between the two modelling spaces is orthogonal because BGVal at M0 layer of the concrete modelling space is defined by using Bigraph theory in the conceptual modelling layer.

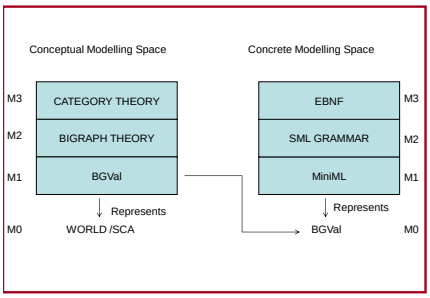

FIGURE 3-2: ORTHOGONAL MODELLING SPACE.

In summary, in this section, we have discussed our choice of using software models as a runtime adaptation mechanism (models at runtime). Also, we have explored the complexity arising out of a high rate and ill-structured order of occurrence of faults that the model at runtime needs to address. We have chosen Bigraphs as a modelling language in the conceptual modelling space. Notice that our thesis explores whether these choices are appropriate to deal with volatility.

3.5 CONCLUSIONS

Our research question seeks to explore the Bigraph expressivity issues that are involved in using Bigraphs to construct a model at runtime, in constructing the mechanisms to support such a model and in the representations of the world and the system that the model must capture. The system that we seek to support in the face of volatility through our proposed Bigraphical model at runtime is a service composition running on a mobile device. To evaluate if we have succeeded in answering the research question we have proposed to construct a proof-of-concept Bigraphical model at runtime and to test if its response times allow it to be in-sync with the real world. To succeed our model's architecture must support the reconfiguration cycle proposed by Manel Fredj (Fredj et al., 2006). In the next chapter, we discuss how to use Bigraphs as a new

language to construct a new architecture- models at runtime- to deal with the problem of volatility in a service composition running on a mobile device.

4 CONSTRUCTING THE ARCHITECTURE FOR A TWO-LAYERED MODEL AT RUNTIME

4.1 INTRODUCTION

In the previous chapter, we discussed our research question and the requirements that our system must fulfill to address that question. To address the research question, in this chapter, we explore a new architecture based on models at runtime (Blair et al., 2009) which we construct using a new language Bigraphs (Milner, 2009) to tackle the problem of volatility (Coulouris, 2012) in a service composition running on a mobile device.

As discussed in Chapter 2, Bigraphs have not been used to construct a model at runtime for such service compositions. We highlight the challenges of writing a model at runtime to support a system running in volatile conditions. We also discuss the advantages to be had by using Bigraphs in such a fashion.

This chapter is organized as follows: In Section 4.2, we discuss a mapping of the observed effects of Chan et al. (Chan et al., 2007a) to the volatility properties of Coulouris et al. (Coulouris, 2012). Next in Section 4.3, we discuss the architecture of our proposed Bigraphical model at runtime. Finally in Section 4.4 we discuss how we program the structure of the WORLD and SCA layers of our Bigraphical model at runtime. We also show how to use the mapping discussed in Section 4.2 to design the reaction rules of our Bigraphical model at runtime. Additionally in Section 4.4, we show how to program a Bigraphical array.

4.2 VOLATILE SERVICE COMPOSITION

In a volatile service composition running on a mobile device, services that are participating might appear and disappear at a high rate. We need a taxonomy that describes the possible fault types that can occur in a service participating in the composition at runtime and the effects that are observed as a result of those faults. These observed effects are at the application level and so can be dealt with using a model at runtime that is causally connected to the application.

We have used a fault taxonomy for web service composition proposed by K.S. May Chan et al.(Chan et al., 2007a) as discussed by us in Chapter 2. From the taxonomy, we have identified those ‘Observed Effects’ of faults that will affect services participating in a service composition running on a mobile device that are triggered by volatility inherent in a ubiquitous computing system (see Table 4-1). Thus, this table extends the taxonomy of Chan et al. by mapping the

faults identified by them to the properties of volatility discussed by Coulouris et al.(Coulouris, 2012).

We use the following definitions from Chan et al (Chan et al., 2007a) for possible classes of fault types because we want to extract out *only* those observed effects that are caused by faults due to volatility. We want our model at runtime to support adaptation to faults that are triggered by volatility *only*. For example, we do not consider development faults because such faults can be triggered even in the absence of volatility.

a) Physical Faults: These are caused by failures in the network medium or failures on the server side. Communication infrastructure exceptions and failures in correct operation of hosting server's middleware are also included.
b) Interaction faults: Services forming part of the composition interact with each other. An interaction fault occurs if a service fails frequently or unacceptably severely.
c) Interaction-content fault: This is a type of interaction fault that includes incorrect service, misunderstood behavior, response error, QoS and SLA faults.

In Table 4-1, we first map the 'Observed Effects' to those of the above classes of fault types that can cause them as discussed in Chan et al.'s taxonomy(Chan et al., 2007a). Next, we map this fault type itself to the type of volatility (Coulouris, 2012) that can trigger it. Notice that besides volatility, other causes may also trigger the faults. Nevertheless, in this thesis we wish to focus only on volatility in order to answer our research question.

We now explain this mapping:

i. According to Chan et al.'s taxonomy, an 'Unresponsive Service' observed effect is caused by one of the following faults as shown in the Table 4-1: Unavailability fault, Timeout fault, Quality of Service (QoS) fault. Next, we map these faults to the volatility properties discussed by Coulouris et al. We identify that the Unavailability and Timeout faults could be caused by volatility resulting from device and communication failures, variation in properties of communication, or destruction of logical communication relationships between software components resident on the devices. A QoS fault could be caused by a slow network because of variation in properties of communication such as bandwidth.
ii. In Chan et al.'s taxonomy, 'Incorrect Results' observed effect is caused by one of the following faults as shown in the Table 4-1: Timeout fault, Quality of Service (QoS) fault. Next, we map these faults to the volatility properties discussed by Coulouris et al. We identify that the Timeout fault could be caused by volatility resulting from device and communication failures, variation in properties of communication, or destruction of

logical communication relationships between software components resident on the devices. A QoS fault could be caused by a slow network because of variation in properties of communication such as bandwidth.

iii. An ‘Incoherent Results’ observed effect in Chan et al.’s taxonomy is caused by Quality of Service (QoS) fault. Next, we map this fault to the volatility properties discussed by Coulouris et al. We identify that the QoS fault could be caused by volatility resulting from a slow network because of variation in properties of communication such as bandwidth.

iv. According to Chan et al.’s taxonomy, a ‘Slow Service’ observed effect is caused by one of the following faults as shown in the Table 4-1: Unavailability fault, Incorrect Order fault, Timeout fault, Quality of Service (QoS) fault. Next, we map this fault to the volatility properties discussed by Coulouris et al. We identify that the Unavailability and Timeout faults could be caused by volatility resulting from device and communication failures, variation in properties of communication, or destruction of logical communication relationships between software components resident on the devices. Incorrect Order and QoS fault could be caused by a slow network because of variation in properties of communication such as bandwidth.

v. An ‘Outdated Results’ observed effect in Chan et al.’s taxonomy is caused by Quality of Service (QoS) fault. Next, we map this fault to the volatility properties discussed by Coulouris et al. We identify that the QoS fault could be caused by volatility resulting from a slow network because of variation in properties of communication such as bandwidth.

TABLE 4-1: MAPPING BETWEEN OBSERVED EFFECTS AND VOLATILITY.

OBSERVED EFFECT (Chan et al., 2007a)	POSSIBLE CLASSES OF FAULT TYPES THAT CAN OCCUR AT RUNTIME AND CAUSE THE OBSERVED EFFECT(Chan et al., 2007a)	TYPES OF VOLATILITY THAT CAN TRIGGER THE FAULT (Coulouris, 2012)
1.Unresponsive Service	Unavailability Fault (Physical Fault), Timeout (Interaction Fault), QoS (Interaction-Content fault)	Device and communication link failure, Variation in properties of communication such as bandwidth, Destruction of logical communication relationships between software components resident on devices
2. Incorrect Results	Timeout (Interaction Fault), QoS(Interaction-Content fault)	Device and communication link failure, Variation in properties of communication such as bandwidth, Destruction of logical communication relationships between software components resident on devices
3.Incoherent Results	QoS(Interaction-Content fault)	Slow network: Variation in properties of communication such as bandwidth
4.Slow Service	Unavailability Fault (Physical Fault), Incorrect Order(Interaction fault), Timeout (Interaction Fault), QoS (Interaction-Content fault)	Device and communication link failure, Variation in properties of communication such as bandwidth, Destruction of logical communication relationships between software components resident on devices
5.Outdated Results	QoS (Interaction-Content fault)	Slow network: Variation in properties of communication such as bandwidth

We assume that a Service Component Architecture (SCA) (Marino and Rowley, 2010) like description of all the services and the service composition is being maintained by a system outside our system boundary. SCA is an architectural specification of a structural composition model for Service –Oriented- Architecture (SOA)(Curbera, 2007). Structural Composition is a model of service composition in SOA which identifies components of a composition that offer services and how those components are connected together (Curbera, 2007), (Marino and Rowley, 2010).

Now that we have mapped 'Observed Effects' of Chan et al. to the types of volatility of Coulouris et al. that can trigger a fault, we identify how volatility also effects the frequency, order, and number of these 'Observed effects' on a system over and above what we might see in a non-volatile system.

i. The 'Observed Effects' on a service due to volatility occur at a high rate at runtime.

ii. In general, we cannot determine in advance the order in which services will suffer from an 'Observed Effect' at runtime- essentially they suffer from an 'Observed Effect' in an ill-structured fashion.

iii. There are an infinite number of possible reconfigurations of a service composition at runtime. These are triggered by a possibly infinite number of applications of the 'Observed Effect' to a composition.

In the next few sections we will describe how we have utilized the above mappings of Table 4-1 in order to design our model at runtime.

4.3 USING MODEL AT RUNTIME AS A CACHE

As discussed earlier, we envisage a service composition running on a mobile device that a shopper is using as she strolls around a shopping mall. This service composition is causally connected to separate entity, which is a Bigraphical model at runtime. This model is also running on the same mobile device on which the composition is running. Also, this service composition is volatile as the participating services appear and disappear at a high rate.

The purpose of our model at runtime is to deal with volatility. We need a way to respond quickly to the high rate of appearances and disappearances of services. As soon as a service malfunctions, we want to substitute it with a new service. Our strategy for a quick response is to cache the pre-fetched location and id of devices which are offering backup of those services that are participating in the composition. This information is cached as a model at runtime. To the best of our knowledge, this thesis is the first to use model at runtime as a cache. Also, when the user moves from one ambient (location) to another, we pre-fetch the location and id of devices from 'nearby' ambients which are offering backup of those services that are participating in the composition. We define 'nearby' as being in the same ambient or in a parent ambient or in a child ambient in the location tree (place graph). With this approach, we can use the model to reason at runtime about which of the backup services are closest to the current location of the user. This is the external environment/context that we wish to model. We call this model the WORLD model. We assume that the devices which are offering substitute services are themselves 'appearing' and 'disappearing' because of volatile wireless connectivity as the user moves around the shopping mall.

We model each of the 'Observed Effect' discussed in the previous section as the state of a service that has developed a fault (See section 4.2 for details). We also model a working service as being in a state called "working". We want to monitor the state of the services comprising the service composition and be able to replace a service whose state has changed from "working" to say "incorrect results". This is an internal view of the system that we want to model. We call it

the Service Component Architecture (SCA) model. Together, WORLD and SCA constitute our model (See Figure 4-1).

As discussed in Chapter 2, the Plato-Graphic Model (PGM) is defined by Birkedal et al. (Birkedal et al., 2006) as combining three Bigraphical Reactive Systems (BRS) with specific roles into one model. The three layers that the three BRS represent are: World, Proxy and Agent. Our implementation of WORLD and SCA layers, using PGM-like ideas, combines two BRS (WORLD, SCA) into one (Figure 4-1). Like PGMs, where the 'World' and 'Agent' layer do not share controls (types of nodes of Bigraphs), our WORLD and SCA layers also don't share their controls.

As discussed earlier, run-time phenomenon are ill-structured as there is no pre-determined order of runtime events, so reaction rules are an appropriate abstraction to model such events. Reaction rules are used to represent the 1) caching of pre-fetched information and user's device's location in the WORLD layer; 2) At the SCA layer; we model the changes in the state of each service participating in the service composition. Notice that the choice of Bigraph as our language offers us an ability to utilise abstractions of the problem space (devices, shops, locations etc) in our models. Utilizing such abstractions of problem space rather than those of solution space has been one of the primary goals of the Model-Driven Engineering community (Schmidt, 2006).

4.3.1 REFERENCE ARCHITECTURE FOR SELF-MANAGEMENT

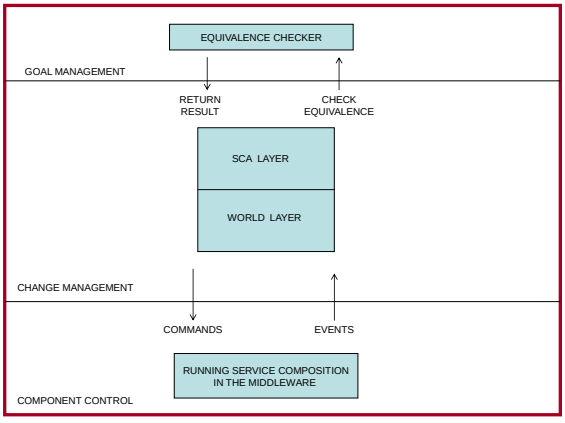

FIGURE 4-1:REFERENCE ARCHITECTURE FOR SELF-MANAGEMENT ADAPTED FROM KRAMER ET AL. (KRAMER AND MAGEE, 2007).

We assume that our Bigraphical model at runtime at the Change Management layer in Figure 4-1 sits on top of a Component Control layer which monitors and feeds events (service malfunctions, movement of user's device) to our model and executes adaptation commands

issued by our model (Figure 4-1). We emphasize that our model at runtime and the service composition are *different* entities.

We have used the three-tier reference architecture proposed by Kramer and Magee (Kramer and Magee, 2007) (Figure 4-1). That is as indicated above, outside our system boundaries, at the lowest Component Control layer, a service composition and it's Service Component Architecture (SCA) (Marino and Rowley, 2010) like description exists.

We assume that at the Component Control layer, there are modules monitoring the composition and report any 'Observed Effects' as events to the Change Management layer above it. Also, we assume that there are modules at the Component Control layer that can support addition, deletion and searching of services. Furthermore, we assume that at this layer, there are event scheduling mechanisms and a supporting module to coordinate between our adaptation strategy and the underlying service composition's exception-handling mechanisms.

Next, within the Change Management layer, we have our system with the two-layered model and associated modules. Our system reacts to the 'Observed Effects' sent as events from the lower layer by sending out the appropriate adaptation commands. These include commands to unbind a faulty service, the command to bind a new service to the composition, and a command to pre-fetch the identity of a device that offers a particular service in a particular ambient. We assume that our system at this layer is causally connected to the service composition.

Finally, at the top-level layer called the 'Goal Management Layer', we assume that an equivalence checker exists outside our system boundary. This checker is used to find if a service is equivalent to a service participating in the composition.

By modelling the environment and system views in separate layers and having reaction rules to represent dynamics in those layers completely captures volatility. We discuss the two views in section 4.4.

4.3.2 MODEL DRIVEN ADAPTATION AT RUNTIME

Model at runtime is a set of techniques that specify an architecture to support self management discussed in the previous section. We deploy architecture similar to Bencomo's and Morin's (Bencomo, 2009, Morin et al., 2008) in order to address the problem of volatility. In Figure 4-2, we assume that the running system forms part of the middleware for service composition. The running system is at layer M0 of MDA. The Plato-graphic model (PGM) like model is depicted as a reference model in Figure 4-2. The reference model (which is a PGM-like model) represents a "damaged" service composition after the application of the reaction rule associated with the event generated by the running system in the middleware. It resides at layer M1 of MDA. The specification model (another PGM-like model) resides at the meta-model layer of

MDA: M2. Notice that the reference model represents a state which does not conform to the specification model. The specification model represents a high level specification of the service composition. The modified model (a PGM-like model) that results from transformation of the reference model using reaction rules conforms to the specification model. The reaction rules used are discussed in Section 4.4. As discussed earlier, we assume that an equivalence checker exists outside our system boundary and it can check if the modified model is equivalent to the specification model. In our implementation using the BPL Tool, we have not used the specification model because we have replaced 'like' service with 'like'.

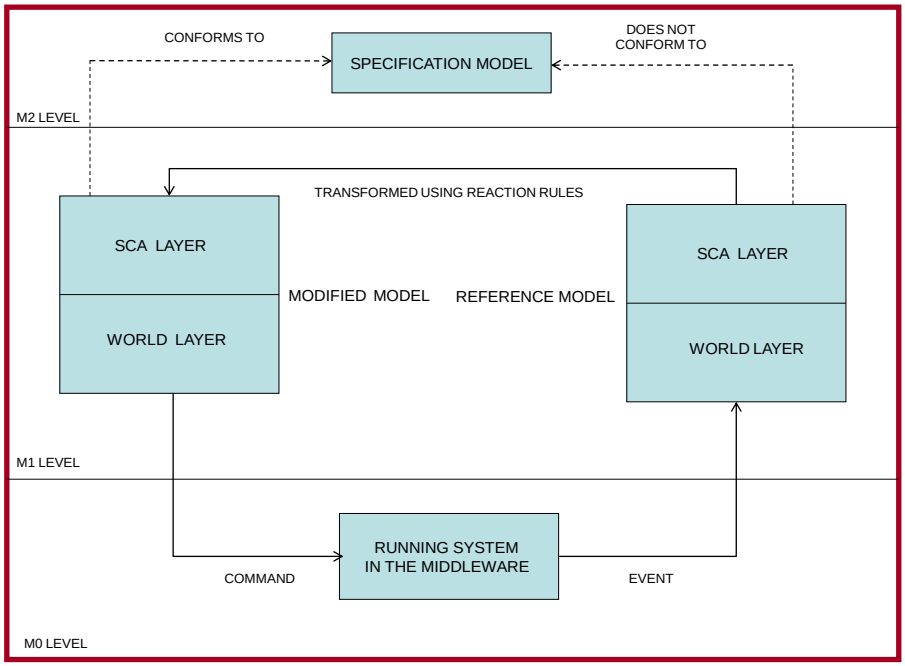

FIGURE 4-2: MODEL DRIVEN ADAPTATION AT RUNTIME.

The commands to effect adaptation are issued by our system after interrogating the model representing the external environment and reflecting on the state of the internal structure of the composition. We discuss the generation of these commands using Bigraphical reaction rules in the next chapter.

4.3.3 DATA FLOW IN OUR MODEL AT RUNTIME

We end this section by giving an operational specification of our model at runtime – its Data Flow Diagram- to describe its desired behaviour. In Figure 4-3, data paths are represented by arrows, ovals represent the processing that the data undergoes and rectangles represent sources and sinks of data. As already discussed, we assume that an event scheduling mechanism exists. Notice that the events are being handled sequentially in our simulations in Chapter 6 rather than concurrently. Each event corresponds to a reaction rule at the appropriate layer and triggers the

firing of a reaction rule at that layer. As discussed above, events at the WORLD layer correspond to the caching of relevant information about the external environment of the service composition (caching and un-caching of device location and movement of user's device). Events at the SCA layer correspond to faults in the internal execution of the service composition. The resulting changes are stored in the data structures representing the WORLD layer or the SCA layer as the case may be. This is represented in Figure 4-3 by the rectangle marked "Changed Service Composition". This information in the data structure is then used by the event processor which is essentially the module where policies that deal with events and system goals are encapsulated to fire a correcting reaction rule resulting in a repaired service composition.

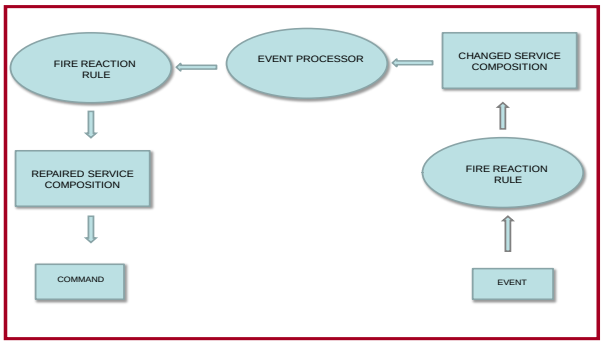

FIGURE 4-3: DATA FLOW DIAGRAM.

Notice that the data flow in our design corresponds to the reconfiguration cycle that we discussed in the previous chapter (Section 3.3.2). The "Fire reaction rule" step in our architecture's data flow diagram in Figure 4-3 corresponds to the Phase 2 of Figure 3-1. The "Event Processor" step in Figure 4-3 corresponds to the Phase 3 of Figure 3-1. And finally, the "Repaired Service Composition" step in Figure 4-3 corresponds to the Phase 4 of Figure 3-1.

4.3.4 SECTION SUMMARY

In this section, we have described the architecture that our Bigraphical model at runtime follows. We first discussed where our two layered model fits within the component control, change management and goal management layers of the reference architecture proposed by Kramer and Magee (Kramer and Magee, 2007). We then described the architecture of our model in terms of Model-Driven-Architecture terminology. Finally, we showed the data flow in our system.

Next, we show how we have constructed our Bigraphical model at runtime keeping the architecture that we have just discussed in mind.

4.4 PROGRAMMING THE STRUCTURE OF WORLD AND SCA LAYERS

The structure of the WORLD and SCA layers represents the static information that our model at runtime captures. The dynamic information is captured by the reaction rules in those layers. These two layers represent two different views of the same system. We now discuss the state of the WORLD and SCA layers, the kinds of nodes each layer contains and how we use MiniML to construct those nodes. Also, recall from Chapters 2 and 3 that MiniML gives us access to SML's control constructs, which are lacking in Bigraphs. One of the goals of developing Bigraphs was to provide such an access to the control constructs (Elsborg, 2009).

From a programming perspective, we view the PGM-like model with the WORLD layer and SCA layer as a data structure that caches and retrieves run-time information. For the WORLD layer, this run-time information includes the location and id of devices whose services comprise the service composition. We also pre-fetch and cache location and id of 'nearby' (same ambient or in a parent ambient or in a child ambient in the location tree) devices that provide backup to those services. For the SCA layer, we cache the structure of the composition and the state of each service participating in that composition. We construct the two layers using the operations described in Chapter 2.

4.4.1 CONSTRUCTING A STATE OF WORLD LAYER

The environment for our system (service composition running on a mobile device) is a smart space. Coulouris et al (Coulouris, 2012) define smart space as "*any physical place with embedded services*". They then describe four types of movements in smart space: Physical mobility, logical mobility, user adds or deletes a device, devices fail.
We have modelled the smart space at the WORLD layer with device ids and their locations. The effects of the four types of movement are modelled through the use of reaction rules at both the WORLD layer and at the SCA layer.

The state of the WORLD layer represents caching of the most up-to date information about elements in the environment pertaining to volatility. This information includes a tree representation of the locations and the devices which may be situated inside the location nodes.

We now discuss the kinds of nodes in the WORLD layer and then the state of a WORLD layer.

4.4.1.1 KINDS OF NODES IN THE WORLD LAYER

There are two main kinds of nodes (called two kinds of controls in Bigraph theory terminology) at the WORLD layer: one representing location and the other devices. We discuss each in turn.

a) Location Nodes: In this thesis, we use the words 'ambient' and 'location' synonymously. In our figures, a location id *label* (as opposed to a Bigraphical control) will be in italics and represented with a string with the letter 'i' followed by a natural number (0, 1, 2, 3 ...). For example, for a location with id '2', the id will be represented in figures as `i2`. The location nodes are constructed by a function called `loc''` provided in Elsborg's code (Elsborg, 2009). The location nodes contain inside them an id node with a string value encapsulated as an atomic Bigraphical node and representing the id of the location. They also contain a site which is a hole in which other Bigraphs could be fitted in. For example, a node representing location `i2` would look as shown in the Figure 4-4.

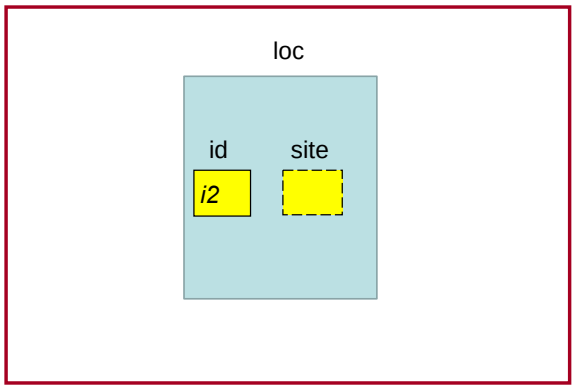

FIGURE 4-4:NODE REPRESENTING A LOCATION WITH ID I2 AND A SITE.

The function `loc''` that constructs a location, takes in the string representation of id as a parameter and constructs the BGVal representation of the location node with a site:

```
fun loc'' n = S.o (loc, S.`|` (S.o (id, i(n)), site))
```

We show the structure constructed by function `loc''` in Figure 4-5.

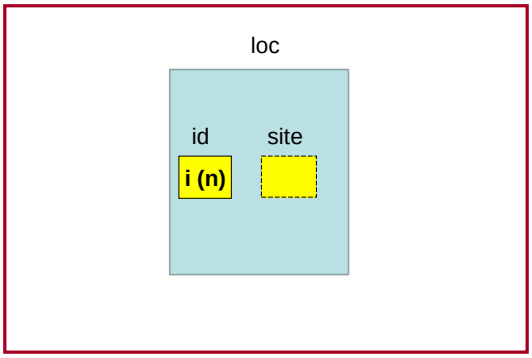

FIGURE 4-5: THE STRUCTURE CONSTRUCTED BY FUNCTION loc''.

We now explain the terms `loc`, `id`, `i (n)` and `site` in the above function and write down the corresponding MiniML code. As discussed in Chapter 2, 'S' is the Sugar module of the BPL Tool:

- `loc` is an active control of zero arity (Elsborg, 2009). In Bigraph theory, an active node can contain child nodes and sites and reaction rules can take place inside them (Milner, 2009). Thus, location nodes can contain nested within them another location node or a device node. The function `active0` of the BPL Tool converts a string into a BGVal representation such that it is an active control of zero arity.

```
val loc = S.active0 "loc"
```

- `id` is a passive control of zero arity (Elsborg, 2009). In Bigraph theory, a passive node can contain child nodes and sites but reaction rules cannot take place inside child nodes (Milner, 2009). The function `passive0` of the BPL Tool converts a string into a BGVal representation such that it is a passive control of zero arity.

```
val id = S.passive0 "id"
```

- `i(n)` is a function that takes in the string representation of id as a parameter and constructs an atomic node of zero arity (Elsborg, 2009). In Bigraph theory, an atomic node cannot contain child nodes and sites (Milner, 2009). The function `atomic0` of the BPL Tool converts a string into a BGVal representation such that it is a atomic node of zero arity.

```
  fun i n = S.atomic0 ("" ^ n)
```

- As discussed in Chapter 2, `site` is a place graph's inner interface. We use the following code from Elsborg's thesis (Elsborg, 2009) to construct a site with BPL Tool's modules and functions. In the following code, the curried function `Per` of BGval module of the BPL Tool constructs a BGVal representing a Bigraphical site.

```
val id_1 = B.Per info (P.id_n 1)

val site = id_1
```

4.4.1.1.1 SIMPLIFIED NOTATION FOR LOCATION NODES

We simplify the notation for location by declaring a value say for location with id `i2` like so:

```
val loc2 = loc'' "2"
```

Therefore location with id `i2` of Figure 4-4 is depicted as shown in Figure 4-6 in a simplified form. Note that *loc2* in italics highlights the fact that it is a label rather than a Bigraphial control. The Bigraphical control in Figure 4-6 labeled `loc2` is `loc`.

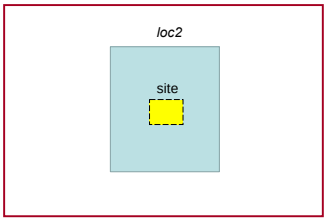

FIGURE 4-6: SIMPLIFIED REPRESENTATION OF LOCATION WITH id i2.

b) Device nodes: For simplicity, we assume that a given device offers a single service only. This one-to-one mapping between devices and services does not reflect any inherent limitation of Bigraphs. However, it does reduce the complexity of the model that we want to use as a starting point for our experiments.

In our Bigraphical model, each device has an id that is a number. The devices are numbered according to the following scheme: Each device id is a decimal number. The number on the left hand side of the decimal point represents the service number that the device offers. The number on the right hand side of the decimal point represents the ordinal number of the device in our model offering this particular service. For example a device with the id "3.4" represents the fourth device in our model that offers service number three. In our model, the device id "0.0" always represents the user's mobile device.

We have designed the device nodes so that their ids can be extracted by using the matching algorithm of the BPL Tool rather than by using the string processing functions of SML. Consider Figure 4-7 were we depict a device with the id `i5.8`. Each device node called `device` (see the Figure 4-7 and Figure 4-8, line 2) is an active node (Milner, 2009) meaning that it can contain child nodes and sites. Moreover, reaction rules can take place inside such active nodes. A `device` node contains another node called `serviceIdNode` (see the Figure 4-7 and Figure 4-8, line 3) which is also an active node. Within the `serviceIdNode` we have an atomic node that

represents the service number being offered by the device. In the Figure 4-7 the device is offering service 5. Recall that an atomic node cannot contain child nodes and sites (Milner, 2009). Also, within the `serviceIdNode` we have an active node called `deviceIdNode` (see the Figure 4-7 and Figure 4-8, line 1). And finally, within this `deviceIdNode` we have an atomic node that represents the ordinal number of the device. In the Figure 4-7, this ordinal number is eight.

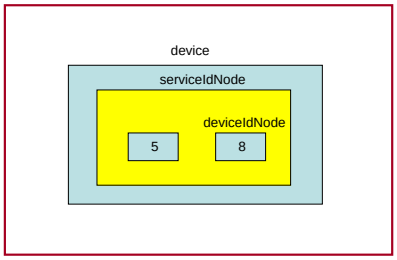

FIGURE 4-7: STRUCTURE OF THE DEVICE BIGRAPH.

```
1   val devIdNode = S.active0 "devIdNode"
2   val device = S.active0 "device"
3   val serviceIdNode = S.active0 "serviceIdNode"
```

FIGURE 4-8: DEFINING `devIdNode, device, serviceIdNode.`

The device nodes are constructed by a function called `constructDevice` that has been written by us. The input parameter of this function is a string called `deviceId`. This `deviceId` must be in the correct decimal number format as discussed above. The function returns a Bigraph representing the `deviceId`.

We now discuss this function's code (Figure 4-9) in detail. In line 2, we check, if "0" has been passed as a parameter to our function. We pass "0" to construct the device with id `0.0`. This device represents the user's device in our model. If the value of the parameter passed to our function is not "0", then the lines 6 to 9 will execute. We extract two strings one in line 8 called `serviceId` representing the service that is being offered by this device and one in line 9 called `deviceId` representing the ordinal number of this device. Finally, these strings are used in lines 11 and 12 to construct a Bigraph representing the device.

```
1  fun constructDevice (deviceId)=
2   if (deviceId = "0") then S.o(device,S.o(serviceIdNode,
3    S.`|`(i("0"),S.o(devIdNode,i("0")))))
4   else
5    let
6     val f = String.tokens (fn x => not((Char.isDigit x)))
7     val identityList = f(deviceId)
8     val serviceId = hd(identityList)
9     val deviceId = hd(tl(identityList))
10   in
11    S.o(device,S.o(serviceIdNode,
12    S.`|`(i(serviceId),S.o(devIdNode,i(deviceId)))))
13   end
```

FIGURE 4-9: FUNCTION `constructdevice`.

To summarize, the function `constructDevice` is used to construct a Bigraph from a string. This Bigraphical device's id can now be extracted using the matching algorithm of the BPL Tool instead of using the string processing functions of SML.

4.4.1.1.2 SIMPLIFIED NOTATION FOR DEVICE NODES

To avoid clutter in our diagrams and discussions, we suffix the letter "i" with the device number of the scheme discussed above for labels of a device. Moreover, to distinguish this label from Bigraphical controls, the label will be in italics. For example `i5.8` in our diagrams and discussion is the eight-device offering service number five. Here, "i" signifies that the number following it is not a 'usual' decimal number but is one that follows our numbering scheme discussed above.

Consider Figure 4-10 where a device with id `i5.8` is shown in a simplified diagram. We will follow the convention of representing device Bigraphs in our diagrams in such a simplified manner in this thesis.

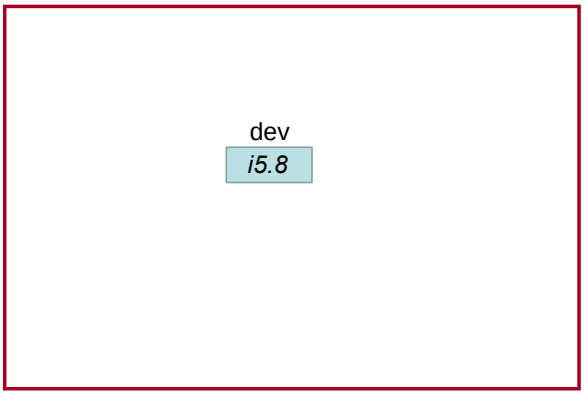

FIGURE 4-10: SIMPLIFIED REPRESENTATION OF DEVICE WITH ID I5.8

4.4.1.2 MODELLING THE ENVIRONMENT VIEW OF EFFECTS OF VOLATILITY WITH THE WORLD LAYER

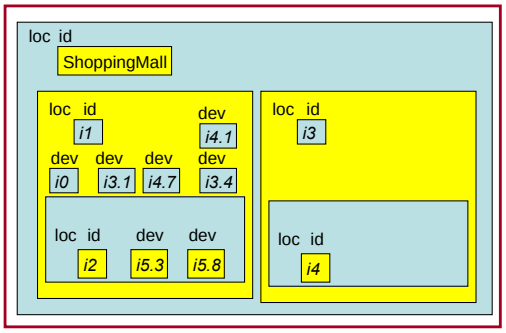

FIGURE 4-11: STATE OF THE WORLD EXPRESSED AS A BIGRAPH.

As shown in Figure 4-11, a state of the WORLD could consist of the location with id `i1` (west wing) and location with id `i3` (east wing) nested inside the mall. The nested ambient (location with id `i2`) within the west wing represents a large clothing store. Similarly, the nested ambient (location with id `i4`) within the east wing represents another competing clothing store. In Figure 4-11, the device with id `i0` is the user's device running the service composition and is in the west wing. This device with id `i0` models the mobile device that the user moves around with in the Shopping Mall. Recall from the scenario in Chapter 1 that the service composition is running on this device. The user is in the west wing so the device with the id `i0` is shown in the

location representing the west wing of the mall. We assume that three services service 3, service 4, and service 5 comprise the service composition. These three services are being offered respectively by devices with id `i3.1`, and `i4.7` on the west wing and device with id `i5.8` in location with id `i2`. We cache the location of the devices whose services are currently participating in the composition. We also cache the location of nearby devices (device `i3.4` for service 3, device `i4.1` for service 4, and device `i5.3` for service 5) that offer backup to each of these services. We define 'nearby' to mean any device in either the current or parent or child ambient. Note that such devices offering backup services might not always exist for us to cache them.

As discussed in Chapter 2, even though the BPL Tool can be used to represent Bigraphs linked together through hyper-graphs, we have not used this capability because the BPL Tool's matching algorithm is not designed to efficiently handle a huge explosion of links that occurs as the size of Bigraph grows (Elsborg, 2009). Thus in the Figure 4-11, the devices with ids `i3.1`, `i4.7`, and `i5.8` whose services are participating in the composition have not been linked together with a hyper-graph although that is permissible in Bigraph theory. Instead, to capture the information that these devices are offering services that are part of the service composition, we will store their ids in a Bigraphical array in our implementation of the model using the BPL Tool.

Because of space constraints, control names `id` and `dev` appear outside their respective node boxes in our figures. On the other hand, control name `loc` appears inside its node box.

The state of WORLD layer of Figure 4-11 expressed in MiniML code is shown in Figure 4-12.

```
val C = S.o(locShoppingMall,S.`|`(S.o(loc1,S.`|`(constructDevice("0.0"),S.`|`
(constructDevice("3.1"),S.`|`( constructDevice("4.7"),S.`|`
(constructDevice("3.4"),S.`|`( constructDevice("4.1"),S.o(loc2,S.`|`
(constructDevice("5.3"), constructDevice("5.8")))))))),S.o(loc3,loc4)))
```

FIGURE 4-12: CONSTRUCTION OF WORLD LAYER.

The dynamics of un-caching and caching and movement of user's device of the WORLD layer are modelled with the reaction rules for that layer. These rules are fired by our functions that encapsulate the adaptation strategy of our system. The first rule models a device being un-cached by our system. In Figure 4-13, we show the reaction rule modelling the un-caching of device `i2.6` from location `i3`. Notice that we will need a separate rule to un-cache each device from each location. In the next chapter, we will show how to tackle this problem by using abstraction by parameterisation techniques.

The second rule models a service being cached. In Figure 4-14, we show the reaction rule modelling the caching of device `i6.7` in location `i4`. Again, notice that we will need a separate rule to cache each new device in each location. We have used abstraction by parameterisation techniques discussed in the next chapter to tackle this problem as well.

Finally, the dynamics of a device moving from one ambient to another can be modelled by application of 'device un-cached' rule in the initial ambient and 'device cached' rule in the final ambient.

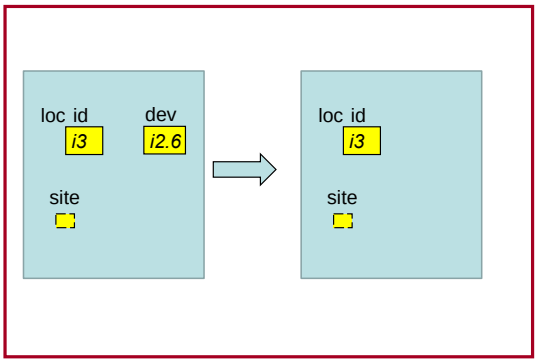

FIGURE 4-13: DEVICE 'UNCACHED' RULE OF THE WORLD.

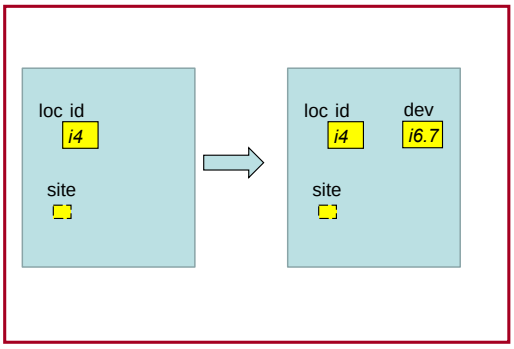

FIGURE 4-14: DEVICE 'CACHED' RULE OF WORLD.

We now discuss each of the movements in smart space and show how these are captured by our two-layered model.

i) Physical mobility: Modelled by the reaction rule at WORLD layer where a device moves from one ambient to another.

ii) Logical mobility: A service might move out of the smart space triggering one of the 'Observed Effects' of K. S. May Chan (Chan et al., 2007a). Each of these 'Observed Effects' is

modelled by a reaction rule at the SCA layer (See next sub-section). Also, device un-cached rule is triggered at the WORLD layer.

iii) User adds or deletes a device: Not part of our scenario. However, the device cached rule can model this type of event.

iv) Devices fail: As above, this triggers one of the 'Observed Effects' which is modelled by a reaction rule at the SCA layer (See next sub-section) and a device un-cached rule at the WORLD layer.

Altogether, each type of movement in a smart space can therefore be captured by our two-layered model. Notice that the *rate of change* of each type is much higher in volatile systems as compared to fixed distributed systems. This means that to keep our model at run time in-sync with the real world, the response times of the functions of our system should be low enough to cope with this higher rate of changes.

4.4.2 CONSTRUCTING A STATE OF SCA LAYER

The second layer of our model at runtime which we call the 'Service Component Architecture' (SCA) layer models the structure of the composition and the state of each service participating in the composition.

Each service is a node containing another node with a string representing the service id and a node with a string representing the state of the service. Also, a service could contain other services within it.

Each one of the Table 4.1's five events associated with the five possible 'observed effects' in a service at the SCA layer is mapped to a reaction rule in our system. A reaction rule changes the state of a service from 'working' which models a properly functioning service to a state named after the fault associated with the reaction rule. The states named after the faults are: 1) Unresponsive, 2) Incorrect result, 3) Incoherent results, 4) Slow service 5) Outdated results.

We now discuss the kinds of nodes in the SCA layer and then the state of an SCA layer.

4.4.2.1 KINDS OF NODES IN THE SCA LAYER

There is only one main kind of node at the SCA layer- the service node. Within the service node, we have a node representing the service id and another node representing the state of the service (See Figure 4-15).

In our figures, a service id label (as opposed to a Bigraphical control) will be in italics and represented with a string with the letter 'i' followed by a natural number (0, 1, 2, 3 ...). For example, for a service with id '7', the id will be represented in figures as `i7`.

A service node is constructed by a function called `constructService` that has been written by us. The service node contains inside it an id node with a string representing the name of the service, a state node with a string representing the name of the state the service is in, and a site that is a hole in which other Bigraphs could be fitted in. This is shown in Figure 4-15 for a service of id `i7` which is in a `working` state. Note that except for Figures 4-15 and 4-16, we will always omit the control name "state" in our diagrams to avoid clutter.

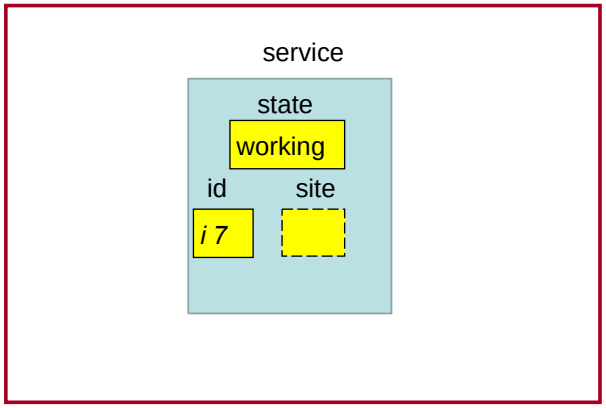

FIGURE 4-15: A SERVICE NODE.

The function `constructService` takes in the string representation of id and string representation of state as parameters and constructs the BGVal representation of a service node with a site:

```
fun constructService(n,someState)=S.o (service,S.`|` (S.o (id,i(n)),S.`|`
(constructServiceState(someState),site)))
```

We show the structure constructed by function `constructService` in the Figure 4-16.

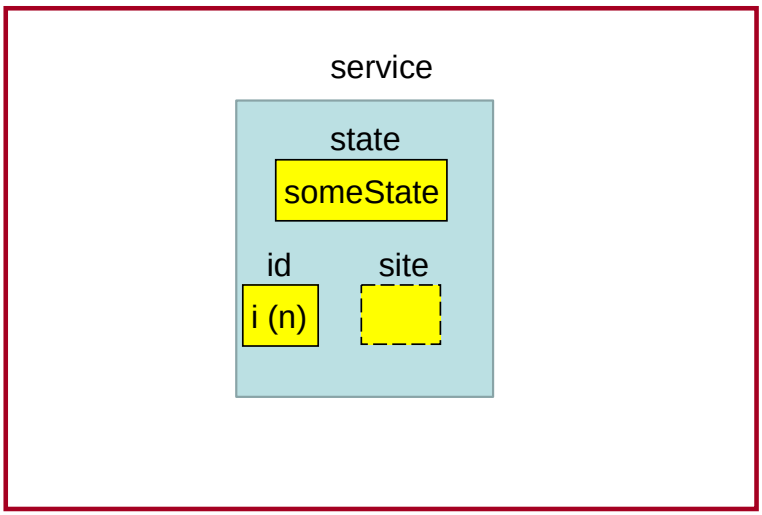

FIGURE 4-16: THE STRUCTURE CONSTRUCTED BY FUNCTION `constructService`.

We have already explained the terms `id`, `i(n)`, and `site` in section 4.4.1.1. We now explain the terms `service`, and `constructServiceState` in the above function:

- `service` is an active control of zero arity. Thus, it can contain other service nodes:

```
val service = S.active0 "service"
```

- `constructServiceState` is a function that takes in the string representation of state as a parameter and constructs an atomic control of arity zero:

```
fun constructServiceState(someState)=S.atomic0 (someState)
```

4.4.2.1.1 SIMPLIFIED NOTATION FOR SERVICE NODES

As with location and device nodes, we can simplify our representation of a service node by declaring a value for a particular service:

```
val service7 = constructService("7","working")
```

Therefore, a service with id 7 of Figure 4-15 is depicted in Figure 4-17 in a simplified form. Note that `service7` in italics highlights the fact that it is a label rather than a Bigraphical control. The Bigraph control in the Figure 4-17 labeled `service7` is `service`.

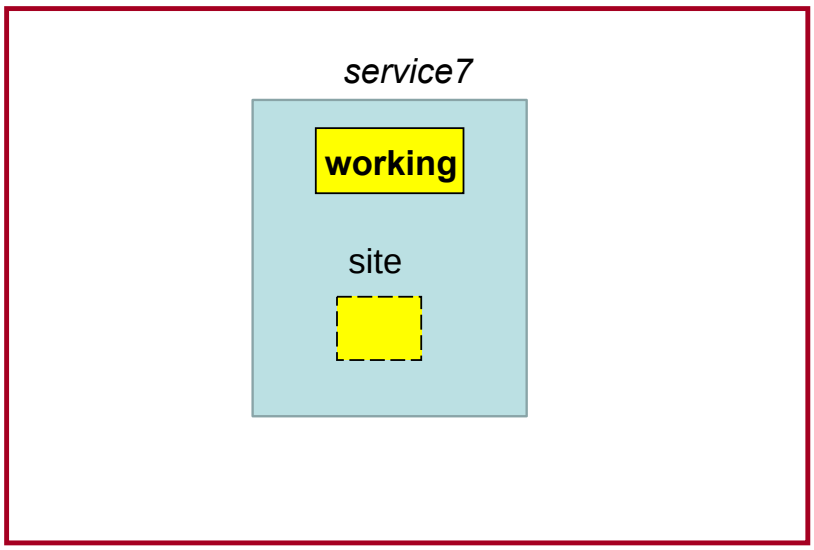

FIGURE 4-17: SIMPLIFIED REPRESENTATION OF SERVICE WITH id i7.

4.4.2.2 MODELLING THE SYSTEM VIEW OF EFFECTS OF VOLATILITY WITH THE SERVICE COMPONENT ARCHITECTURE (SCA) LAYER

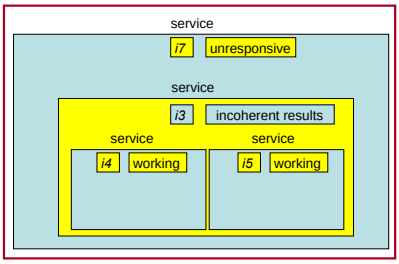

FIGURE 4-18: STATE OF SCA LAYER.

The states at the SCA layer are modelled as nodes in our bigraph model (Figure 4-18). These states are based on faults discussed by Chan et al.(Chan et al., 2007a). In addition, we have a state called `not working` which models any other fault that we have not captured. In Figure 4-18, service `i7` is a bigraph modelling a composite service. The state of service `i7` is `unresponsive`. It consists of three services: service `i3` which is in the state `Incoherent results` and services `i4` and `i5` which are in the `working` state. Notice that we assume that the monitoring system- which is outside our system boundary- 'knows' that service `i4` and service

i5 on their own are working fine and that the problem is with service i3 on which service i7 depends. We have captured this 'knowledge' in our model.

Note that the control name service appears outside its node box.

To conclude, the state of SCA layer shown in Figure 4-18 as expressed in MiniML code is shown in Figure 4-19. Notice that the sites of services i4 and i5 have been filled-up with barren roots. In Bigraphs, barren roots are those that do not have any children. This root is constructed using the following code from Elsborg (Elsborg, 2009) where <-> is a BGVal denoting a Bigraphical root and is defined in the Sugar module of the BPL Tool:

```
val barren = S. <->
```

```
val L = S.o(constructService("7", "unresponsive"),S.o(constructService( "3",
 "incoherent results"),S.`|`(S.o(constructService("4",
 "working"),barren),S.o(constructService( "5", "working"),barren))))
```

FIGURE 4-19: CONSTRUCTION OF A STATE OF SCA LAYER.

4.4.2.3 EXPRESSIVENESS OF OUR SCA MODEL

Our model at the SCA layer can express all possible types of service compositions. We now discuss how we have used the types of component compositions to derive our types of service composition.

As discussed earlier, we have assumed that an SCA like description of the actual service composition exists in the layer below the layer where our system exists. Both SCA and Architecture Description Languages (ADLs) like Darwin (Magee et al., 1995) represent component composition using a service ('provides') interface and a 'requires' interface as shown in the Figure 4-20.

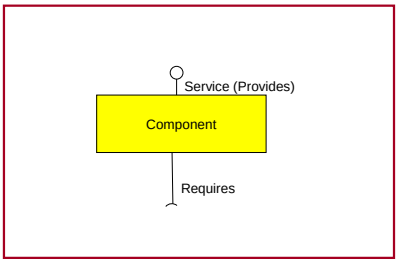

FIGURE 4-20: COMPONENT INTERFACES (SOMMERVILLE, 2011).

There are three kinds of component compositions (Sommerville, 2011):

1) Sequential composition: In such a composition, firstly service, say S1 offered by component say C1 is called by the system. Then, the result is used in a call to another service say S2 offered by a component say C2. This is shown in Figure 4-21.

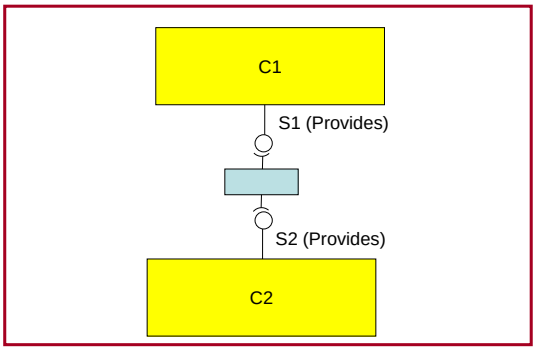

FIGURE 4-21:SEQUENTIAL COMPONENT COMPOSITION (SOMMERVILLE, 2011).

2) Hierarchical composition: Here, service S1 offered by component C1 is called by the system. S1 in turn calls service S2 offered by component C2. See Figure 4-22.

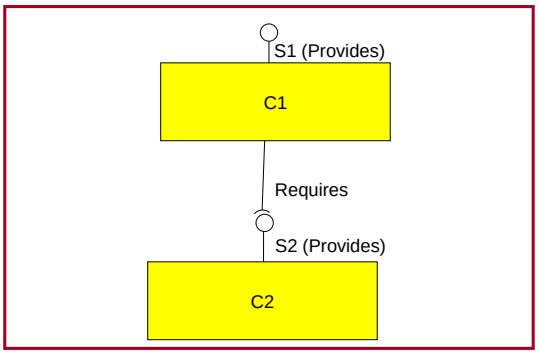

FIGURE 4-22:HIERARCHICAL COMPONENT COMPOSITION (SOMMERVILLE, 2011).

3) Additive composition: In this case, an external interface encapsulates two independent services S1 and S2 respectively offered by components C1 and C2. This common interface is used by the system to call the two services S1 and S2. S1 and S2 do not call each other. See Figure 4-23.

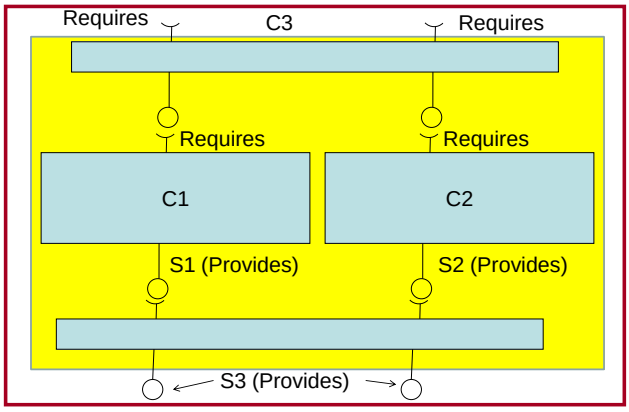

FIGURE 4-23: ADDITIVE COMPONENT COMPOSITION (SOMMERVILLE, 2011)

As discussed earlier, the BPL Tool does not support the use of links. So, we abstract out in our model only the 'provides' interface of a component that specifies what service is provided. Our corresponding service composition models of the above component compositions are discussed now:

1) Sequential composition: This corresponds to a call made to service `i1` by our system (user's mobile device) and using the results in another call to service `i2`. The Bigraph model of such a composition is shown in Figure 4-24. Note that we encode order within our SML code through our numbering system- the lowered numbered service comes first.

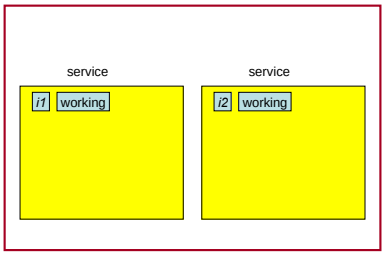

FIGURE 4-24: BIGRAPH MODEL OF SEQUENTIAL SERVICE COMPOSITION.

2) Hierarchical composition: The system (user's mobile device) calls service `i2` which in turn calls service `i1`. This composition is depicted in our Bigraph model as shown in the Figure 4-25.

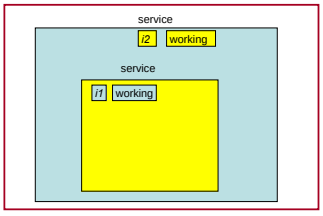

FIGURE 4-25: BIGRAPH MODEL OF HIERARCHICAL SERVICE COMPOSITION.

3) Additive composition: The external interface service `i3` is called by the system (user's mobile device). The device that offers service i3 then calls a device offering service `i1` and another device offering service `i2`. This type of composition is depicted in our Bigraph model as shown in the Figure 4-26.

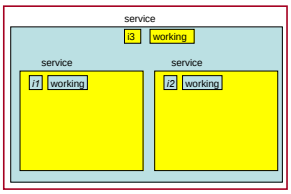

FIGURE 4-26: BIGRAPH MODEL OF ADDITIVE SERVICE COMPOSITION.

Thus, we see that our model at the SCA layer can express all types of service compositions identified by Sommerville (Sommerville, 2011).

4.4.2.4 BRIDGING THE GAP BETWEEN ARCHITECTURE AND REQUIREMENTS

Enhancing the architecture of a system by adding states of each of its components as an extra element is important to bridge the gap between architecture and requirements of a system (Hirsch et al., 2006).

As discussed earlier, our requirement is that our system responds quickly to a service malfunction by substituting it with an equivalent service. We deal with this requirement by caching the location and id of devices at WORLD layer and caching the structure of composition along with the state of each service at the SCA layer.

We cache the following 'Observed Effects' (Chan et al., 2007a) as states in our model at the SCA: 1) Unresponsive service, 2) Incorrect result, 3) Incoherent results, 4) Slow service 5) Outdated results. We have chosen these effects as they result from faults at runtime rather than those at development time. Caching the observed effect as states in the architectural model enables us to express the difference between a working and a malfunctioning composition even though it might have an un-altered structure. Within our system, the requirement of responding quickly is met by identifying a malfunctioning service through its state and replacing it with an equivalent service. Notice that a system outside our system boundary needs to send us the event corresponding to the change in the state of the service. Thus, this external system might still delay the overall response of the system. Since this issue of the external system being slow is beyond the scope of our thesis, we assume that *given* that such an external system is quick, how low are the response times of our system.

4.4.2.5 CAPTURING ALL POSSIBLE OBSERVED EFFECTS ON SERVICES THROUGH REACTION RULES

Our goal has been to develop the smallest number of reaction rules that capture all the effects of volatility on a service composition running on a mobile device. However, for the purpose of building our system we consider only those cases where the number of services and the structure of composition remains the same and only one service in the composition develops a fault at one time. As a result, we do not have reaction rules to deal with the change in the number of services or the structure of the composition.

As discussed in section 4.2, we have used a fault taxonomy for web service composition proposed by K.S. May Chan et al.(Chan et al., 2007a). From the taxonomy, we have identified those 'Observed Effects' of faults that will affect services participating in a service composition running on a mobile device that are triggered by volatility inherent in a ubiquitous computing system (see Table 4-2 on the following pages). We model each of these observed effects as an event sent to our system by another system outside our system boundary. Each of these events has a corresponding state of service named after it. However we can create additional states because the function that encapsulates the reaction rule to change the state is parameterized (See next chapter). Notice therefore that a fault for the service composition is an event for our system. We have written reaction rules that change the state of a service from 'Working' to one of the aforementioned states. We now extend Table 4-1 presented in section 4.2 with an additional column of our reaction rules in Table 4-2 and discuss these reaction rules:

i. Service state at the SCA layer changes from 'Working' to 'Unresponsive Service': For a service of id `i5` the reaction rule will be as show in the Figure 4-27:

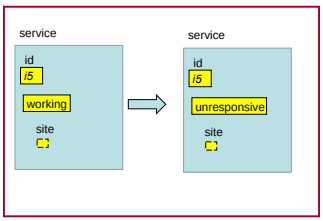

FIGURE 4-27: STATE CHANGE OF A SERVICE FROM WORKING TO UNRESPONSIVE.

ii. Service state at the SCA layer changes from 'Working' to 'Incorrect Results': For a service of id i5 the reaction rule will be as show in the Figure 4-28:

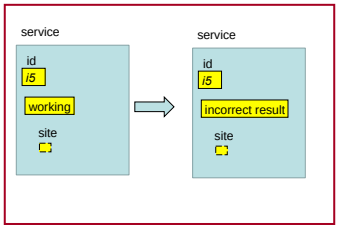

FIGURE 4-28: STATE CHANGE OF A SERVICE FROM WORKING TO INCORRECT RESULTS.

iii. Service state at the SCA layer changes from 'Working' to 'Incoherent Results': For a service of id `i5` the reaction rule will be as show in the Figure 4-29:

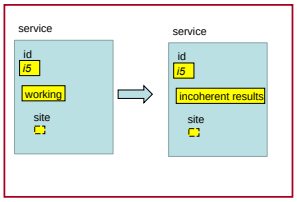

FIGURE 4-29: STATE CHANGE OF A SERVICE FROM WORKING TO INCOHERENT RESULTS.

iv. Service state at the SCA layer changes from 'Working' to 'Slow Service': For a service of id `i5` the reaction rule will be as show in the Figure 4-30:

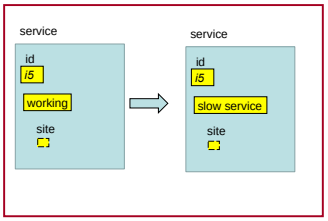

FIGURE 4-30: STATE CHANGE OF A SERVICE FROM WORKING TO SLOW SERVICE.

v. Service state at the SCA layer changes from 'Working' to 'Outdated Results': For a service of id `i5` the reaction rule will be as show in the Figure 4-31:

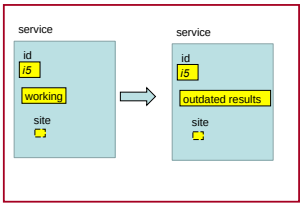

FIGURE 4-31: STATE CHANGE OF A SERVICE FROM WORKING TO OUTDATED RESULTS.

We add an additional column titled 'Reaction rule at the SCA layer' to Table 4-1 to produce Table 4-2:

TABLE 4-2: MAPPING BETWEEN REACTION RULES AND VOLATILTY.

REACTION RULE AT THE SCA LAYER	OBSERVED EFFECT (Chan et al., 2007a)	POSSIBLE FAULT TYPES THAT CAN OCCUR AT RUNTIME AND CAUSE THE OBSERVED EFFECT(Chan et al., 2007a)	TYPES OF VOLATILITY THAT CAN TRIGGER THE FAULT (Coulouris, 2012)
1. Service state at the SCA layer changes from 'Working' to 'Unresponsive Service'	Unresponsive Service	Unavailability Fault (Physical Fault), Timeout (Interaction Fault), QoS (Interaction-Content fault)	Device and communication link failure, Variation in properties of communication such as bandwidth, Destruction of logical communication relationships between software components resident on devices
2.Service state at the SCA layer changes from 'Working' to 'Incorrect Results'	Incorrect Results	Timeout (Interaction Fault), QoS(Interaction-Content fault)	Device and communication link failure, Variation in properties of communication such as bandwidth, Destruction of logical communication relationships between software components resident on devices
3. Service state at the SCA layer changes from 'Working' to 'Incoherent Results'	Incoherent Results	QoS(Interaction-Content fault)	Slow network: Variation in properties of communication such as bandwidth
4. Service state at the SCA layer changes from 'Working' to 'Slow Service'	Slow Service	Unavailability Fault (Physical Fault), Incorrect Order(Interaction fault), Timeout (Interaction Fault), QoS (Interaction-Content fault)	Device and communication link failure, Variation in properties of communication such as bandwidth, Destruction of logical communication relationships between software components resident on devices
5. Service state at the SCA layer changes from 'Working' to 'Outdated Results'	Outdated Results	QoS (Interaction-Content fault)	Slow network: Variation in properties of communication such as bandwidth

We can also generate a reaction rule that changes any malfunctioning state of a service back into the 'Working' state. This reaction rule corresponds to an adaptation command given out by our system that binds a replacement service to the composition.

We have now seen that our reaction rules capture faults extracted from Chan's taxonomy (Chan et al., 2007a) that can occur in a service composition running on a mobile device because of volatility.

4.4.3 A BIGRAPHICAL ARRAY TO SUPPORT SERVICE COMPOSITION

We wish to store those devices whose services are participating in the composition in a Bigraphical array. This is needed to inform the adaptation that will be required if one of the devices that are participating develops a fault and needs to be replaced by another device.

4.4.3.1 KINDS OF NODES IN THE BIGRAPHICAL ARRAY

There are two main kinds of nodes (called two kinds of controls in Bigraph theory terminology) in the Bigaphical array: one representing the array itself and the other representing those devices which are contained within the Bigraphical array. We have already discussed the device node in previous sections. We now discuss our Bigraphical array which we call `compositionDevices`.

`compositionDevices` is an active control of zero arity (Elsborg, 2009). As discussed earlier, in Bigraph theory, an active node can contain child nodes and sites and reaction rules can take place inside them (Milner, 2009). Thus, a `compositionDevices` node can contain nested within it a device node. The function `active0` of the BPL Tool converts a string into a BGVal representation such that it is an active control of zero arity.

```
val compositionDevices = S.active0 "compositionDevices"
```

We use this `compositionDevices` node to construct a `barren` Bigraph:

```
val A = S.o (compositionDevices, barren)
```

In Bigraph theory, a node or root is barren if it has no children (Milner, 2009) . Finally, we use BPL Tool's `makePlato` function to construct a parallel product of the WORLD layer, SCA layer and the Bigraphical array:

```
val system0 = makePlato(C,L,A)
```

In the code above, `C` is the Bigraph representing the WORLD layer, `L` is a Bigraph representing the SCA layer, and `A` is the Bigraph representing the Bigraphical array called `compositionDevices`.

To add devices to the Bigraphical array, we have written a function called `deviceJoinsComposition` (see next chapter). Similarly, to remove a device from the Bigraphical array, we have written a function called `deviceLeavesComposition` (see next

chapter). We have also written a function called `newFindParticipatingDevice` to find the id of a device whose service is part of the composition (see next chapter).

4.4.3.2 STATE OF THE BIGRAPHICAL ARRAY

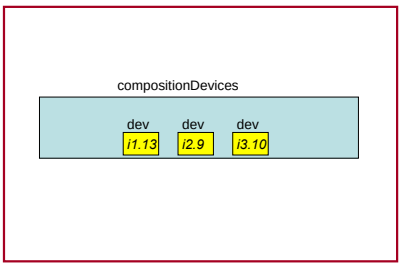

FIGURE4-32: STATE OF THE BIGRAPHICAL ARRAY.

As shown in the Figure 4-32, a state of the Bigraphical array `compositionDevices` could consist of three devices whose services are participating in the composition. In the figure we assume that service 1, service 2 and service 3 are participating in the composition. The devices which are currently being used for this composition are devices with id `i1.13`, `i2.9` and `i3.10` as seen in the figure. Notice that we are using a simplified diagram for each device to avoid clutter. To conclude, the state of Bigraph of Figure 4-32 as expressed in MiniML code is shown in Figure 4-33.

```
val A =

S.o(compositionDevices,S.`|`(S.o(device,S.o(serviceIdNode,S.`|`(3,S.o(devIdNod
e,10)))),S.`|`(S.o(device,S.o(serviceIdNode,S.`|`(1,S.o(devIdNode,13)))),
S.o(device,S.o(serviceIdNode,S.`|`(2,S.o(devIdNode,9)))))))
```

FIGURE 4-33: CONSTRUCTION OF A STATE OF BIGRAPHICAL ARRAY.

4.4.4 SECTION SUMMARY

In this section, we have shown how to capture the relevant information at runtime from two points of view- the external environment (locations in the WORLD layer) and the internal structure of the service composition (services in the SCA layer). By using the same Bigraph constructed out of a parallel product of the Bigraph (See chapter 2) representing the WORLD layer and another Bigraph representing the SCA layer, we have shown a way to combine two models of runtime into one.

We first described how our WORLD layer captures the environment view of the effects of volatility by caching device ids and locations and having reaction rules that model the caching

and un-caching of devices. We also discussed how we could simulate the movement of the user's device by triggering first the un-caching and then the caching of device rules. Next, we discussed how our SCA layer can capture sequential, hierarchical, and additive service compositions. We showed that our use of state whilst describing the service composition helps to bridge the gap between requirements of volatility and the architecture. We also showed that our reaction rules at the SCA layer completely capture all the 'Observed Effects' of faults in a service composition due to volatility. Finally, we discussed the construction of a Bigraphical array to store the identity of those devices whose services are participating in the service composition.

4.5 CONCLUSIONS

We have shown in this chapter, how to use a new language -Bigraphs- to construct a new architecture - models at runtime - to deal with the problem of volatility in a service composition. We use our two-layered model at runtime to cache pre-fetched information about the environment (WORLD layer) and the system (SCA layer). This information is stored before a fault occurs – in particular the id of alternative back-up devices and their locations are stored in the WORLD layer of the model and in this sense the information is pre-fetched. This information can be used to quickly replace a malfunctioning service with an equivalent service nearby. The SCA layer captures all the observed effects of faults on a service composition through its reaction rules. It also captures all possible types of service composition and considers state of a service as part of architecture thereby bridging the gap between the requirements and architecture of a system. Moreover, reaction rules at both the WORLD and SCA layer capture all kinds of appearances and disappearances of services in a smart space. We have therefore shown a way to capture two views of a system in one runtime model. In conclusion, we have demonstrated how to leverage abstractions provided by Bigraphs to use a model at runtime as a cache to deal with volatile service composition running on a mobile device.

5 USING THE BPL TOOL TO IMPLEMENT A TWO-LAYERED MODEL AT RUNTIME

5.1 INTRODUCTION

As discussed in Chapter 2, to the best of our knowledge, our work is the first to have implemented a Bigraphical model at runtime using the BPL Tool. This chapter serves as a proof-of-concept that Bigraphs can be used as a language in the conceptual modelling space (See Chapter 3) to express a two-layered model at runtime for managing ubiquitous computing volatility. As discussed in Chapter 3, we implement the conceptual model expressed in Bigraphs with a concrete model expressed in MiniML. Terms representing Bigraphs can be written as MiniML constructs by using the BPL Tool (ITU, 2007a, Elsborg, 2009). We have built our code on top of Ebbe Elsborg's code (Elsborg, 2009).

In this chapter, we discuss ways in which we have organized our MiniML code such that it could be independently re-implemented if the reader so wished. A Plato-Graphic model (PGM) (Birkedal et al., 2006) like idea has been used by us to model the structure of the WORLD and SCA layer. Thus, we present a way to combine two views (environment and system) into the same model. We also show how to parameterize the reaction rules so that the matching algorithm (Birkedal et al., 2007) of the tool returns a single match giving us the ability to dynamically query and modify the model at runtime. The same parameterization techniques are also used by us to generate infinitely many reaction rules intensionally.

We have organized the chapter as follows: In section 5.2, we discuss the approach that we have taken to implement our system. Then in section 5.3, we discuss the functions that we have written to access and modify the WORLD and SCA layers. Finally in section 5.4, we discuss functions that we have written to encapsulate adaptation logic as well as scripting functions that we have used to run our implementation. In the next section, we give an informal commentary on our implementation.

5.2 IMPLEMENTATION APPROACH

Our system's implementation successfully utilises Bigraphs as a language to construct a model at runtime. We now discuss some of the simplifying assumptions that we have made about our system's boundaries, those features of Bigraphs that we have not used, and how our system

responds to external events by issuing commands to components outside the system boundary and by making internal changes.

5.2.1 SYSTEM BOUNDARY

We have kept the implementation simple enough for us to focus on exploring the most appropriate way to construct a model at runtime with Bigraphs. Our system gets a stream of events and responds by outputting the appropriate commands. Everything else is outside our system boundary. This means we assume that there is a software layer underneath our system at the Component Control layer (See section 4.3.1) which monitors the service composition and reports all the relevant events to our system.

As discussed in Chapter 4, we assume that a Service Component Architecture (SCA) (Marino and Rowley, 2010) like description of all the services and the service composition is being maintained by a system outside our system boundary in the Component Control layer. Note therefore that our model at runtime and the service composition are *different* entities and that both are causally connected. We also assume that if the rate of incoming events exceeds the response rate of our system, then there is an event scheduling mechanism outside our system boundary in the Component Control layer that deals with it appropriately. We are handling the events sequentially rather than concurrently. Again, we assume that there is a mechanism outside our system boundary in the Component Control layer which applies our adaptation commands to the service composition. Note that we un-cache a device from our model at runtime (See Chapter 4) not just when volatility causes a service malfunction on that device but also in case of malfunction caused due to any other reason. We assume that there is a supporting module in the Component Control layer that exists to coordinate between our adaptation strategy (which is essentially to rebind to any other 'equivalent' service that is being offered by one of the 'nearby' devices whose location has been pre-fetched by us) and the underlying service composition's exception-handling mechanisms. Our volatility handling mechanisms are at a higher level of abstraction (modelling level at the Change Management layer) than those exception handling mechanisms (service composition language at the Component Control layer). Note too that even if the code running the service composition is verifiably correct, the composition will still malfunction because of volatility. Furthermore, we assume that there is an equivalence checker outside the system boundary at the Goal Management layer (See section 4.3.1) that decides if two services are equivalent.

5.2.2 UNUSED FEATURES OF BIGRAPHS

As discussed in Chapter 2, because of the limitations of the matching algorithm of BPL Tool, we do not use Bigraph theory's hyper-graphs in our model. Thus, instead of connecting all the devices (that are participating through their services in the service composition) with hyper-graphs in the WORLD layer, we use a Bigraphical array which we discussed in Chapter 4 to store the device Ids of such devices. This Bigraphcal array helps us write code that can distinguish between participating devices and those that are only offering back-up services but have nevertheless been cached as part of our pre-fetching strategy.

5.2.3 EVENTS AND COMMANDS IN THE SYSTEM

In our system, most events (modelled by a reaction rule- see Table 5.1) that need to be processed are associated with corresponding commands. Bigraphs give us a perfect abstraction to model events in the world and associate policies and actions (commands) with those events. Note that although we know how to respond to each event, we do not know the order in which they will occur as runtime phenomenon are ill-structured in that there is no pre-determined order for their occurrence. As reaction rules can be fired in any order they are an appropriate abstraction. As discussed in Chapter 4, we always represent the user's device as 'devi0' in a diagram. This is the device that we assume is running the service composition. Both commands and events are also associated with internal actions taken by our system. We categorise events as follows:

1. Five events associated with the five possible faults in a service at the SCA layer. As described in Chapter 4, we model the following events (observed effects) (Chan et al., 2007a) resulting from faults in a running service composition: 1) Unresponsive service, 2) Incorrect result, 3) Incoherent results, 4) Slow service 5) Outdated results. We have chosen these effects as they result from faults at runtime rather than those at development time. Internally, our system updates the SCA layer to reflect the observed effects.
2. The event where 'devi0' moves from one ambient to another. Internally, we update the WORLD layer to reflect this movement.
3. The event where we receive back results from the software layers underneath about the location and identity of a device offering a particular service. Internally, we cache this information in our model.

We categorise commands that our system outputs to a system outside our system boundary (an Android machine- see next chapter) as follows:

1. The command to unbind a faulty service from the service composition. Internally, we also un-cache the service's device in the WORLD layer and change the state of the service as appropriate to one of the five fault states in the SCA layer.
2. The command to bind a new service to the composition. Internally, this service's device and its location have already been pre-fetched in the cache. Also, we change the state of service back to 'working' (See Chapter 4).
3. The command to pre-fetch the identity of a device that offers a particular service in a particular ambient.

We show the mapping of each event to the set of commands in Table 5-1.

TABLE 5-1: MAPPING INPUT EVENTS TO OUTPUT COMMANDS.

EVENT	OUTPUT COMMAND AND INTERNAL ACTIONS
(i) A service suffers one of the five faults: 1) Unresponsive service, 2) Incorrect result, 3) Incoherent results, 4) Slow service 5) Outdated results. Internally, we update the SCA layer.	1. Unbind the faulty service from the service composition. Internally, our system also un-caches the service's device in the WORLD layer and changes the state of the service as appropriate to one of the five fault states in the SCA layer. 2. Send out a command to bind a new service to the composition. Internally, this service's device and its location have already been pre-fetched in the cache by our system. Also, our system changes the state of service back to 'working' (See Chapter 3). 3. Send out a command to pre-fetch the identity of a device that offers an alternative backup service in the new ambient.
(ii) 'devi0' moves from one ambient to another. Internally, we update the WORLD layer.	Send out a command to pre-fetch the identity of all devices that offer those services which are participating in the service composition but have not been cached in the WORLD layer for the new ambient.
(iii) We receive back results from the software layers underneath about the location and identity of a device offering a particular service. Internally, we cache this information in our model. There is no need to output any command.	

In the implementation, we replace one service with another equivalent service. As discussed earlier, we assume that there is an equivalence checker outside our system boundary- note that developing such a checker is not a trivial problem. However, developing such a checker is outside the scope of the thesis.

The reaction rule associated with an event is run through a matching algorithm (Birkedal et al., 2007). This algorithm determines for a given Bigraph and reaction rule whether and how the reaction rule can be applied to rewrite the Bigraph. The algorithm outputs a set of possible Bigraphs that result from the application of the reaction rule. This algorithm has been implemented in MiniML by the BPL Tool (ITU, 2011) and we build our code on top of this algorithm.

This systematic mapping between Events and Commands discussed above makes our task of programming our model at runtime less complex.

5.2.4 SECTION SUMMARY

Our implementation is a system that responds to external events by issuing commands and changing its internal state. In the next few sections, we give a detailed account of the implementation.

5.3 FUNCTIONS TO MODIFY/ACCESS THE WORLD/SCA LAYERS

As discussed in Chapter 2, one of the goals of designing a model at runtime is to develop a mechanism to use the information stored in the model to take decisions at runtime. Secondly, because the models are causally connected to the running system and the world, we need to be able to use the causal connection to modify the model to reflect any changes in the running system and the world. For meeting both these goals, we have written functions to modify/access the WORLD/SCA layers. We view the functions that we have written as algorithms to access or modify the WORLD/SCA data structures. Matching algorithm (Birkedal et al., 2007) of BPL Tool is the core around which these functions are organized.

Consider a reaction rule R->R' where R is the redex Bigraph and R' is the reactum Bigraph. Let Bigraph A be a bigraph:

$$A = C \circ (R \otimes id_Z) \circ d$$

Bigraph A, on application of the reaction rule can be re-written as:

$$A = C \circ (R` \otimes id_Z) \circ d.$$

The operator $\otimes$ is called a tensor product. It constructs a larger Bigraph by placing two smaller Bigraphs that do not share names (See Chapter 2) side-by-side. The operator 'o' denotes composition of Bigraphs (Birkedal et al., 2007), 'C' is the context, 'R' is the reactum, 'R`' is the redex,' d' is a parameter and 'Z' is the set of names of 'd'. The parameter 'd' is a Bigraph that

fits into the sites (don't cares) of reaction rules. id_Z is the identity function for composition that '*allows a name 'Z' to be "passed through" the redex and be attached to something in the context 'C'*' (Birkedal et al., 2007).The sites of a reaction rule are don't cares in the sense that it does not matter, which, Bigraph is in the same place as the site, for a reaction rule to be applied. We pass the larger Bigraph A and redex R to the matching algorithm which must find C, d and Z such that

$$A = C \circ (R \otimes id_Z) \circ d$$

holds. Since, in our implementation, we do not use links; the matching algorithm only returns C and d.

As an example of how the matching algorithm works, consider the reaction rule shown in Figure 5-1:

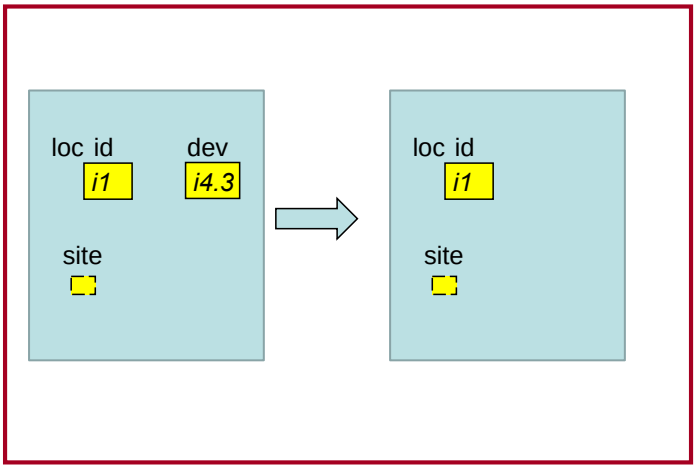

FIGURE 5-1: DEVICE UN-CACHING REACTION RULE.

This reaction rule models the un-caching of device `i4.3` from location `i1`. Any Bigraph returned by the matching algorithm as the parameter 'd' is fitted in the sites of the reaction rule. We apply this reaction rule to the Bigraph in Figure 5-2 which shows a state of the WORLD layer.

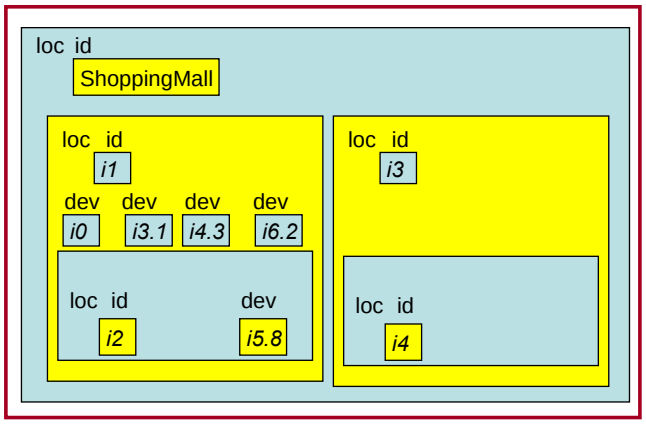

FIGURE 5-2: INITIAL STATE OF THE WORLD BIGRAPH.

The redex (left hand side) of this reaction rule is passed to the matching algorithm along with the Bigraph of Figure 5-2. The context C and parameter d that the matching algorithm returns are shown respectively in Figure 5-3 and Figure 5-4:

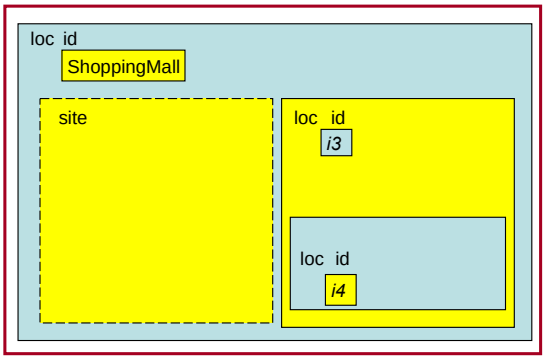

FIGURE 5-3: CONTEXT RETURNED BY MATCHING ALGORITHM.

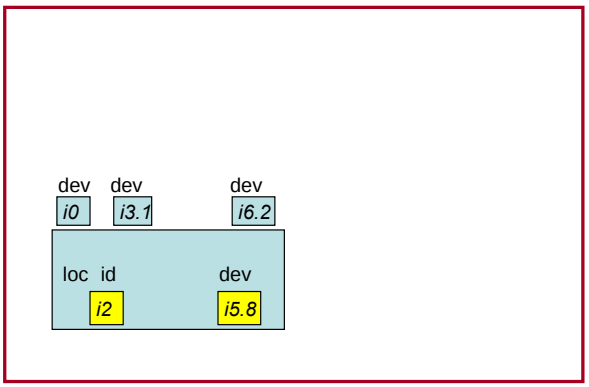

FIGURE 5-4: PARAMETER RETURNED BY MATCHING ALGORITHM.

The context ‘C’ is a Bigraph with a site into which the reactum of the Bigraph can fit in. Also, the parameter returned consists of a Bigraph that goes into the site of the redex of the reaction rule shown in Figure 5-1.

After the application of the reaction rule, the Bigraph of Figure 5-2 is re-written into Bigraph shown in Figure 5-5 with the device `i4.3` having been un-cached.

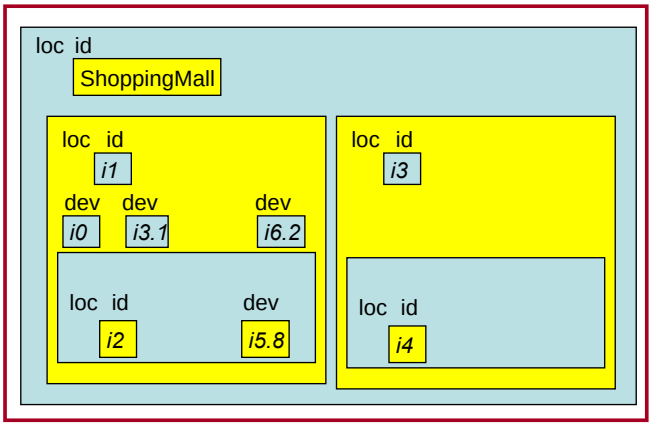

FIGURE 5-5: REWRITTEN STATE OF THE WORLD.

Next, in sections 5.3.1 and 5.3.2, we discuss two classes of functions that have used the matching algorithm:

- Functions that modify the model.
- Functions that access information from the model.

5.3.1 FUNCTIONS THAT MODIFY THE MODEL

These functions encapsulate the reaction rules of the model. Note that for concrete implementation, we use the function abstraction of MiniML. For conceptual modelling of domain knowledge, reaction rules are appropriate to express the semantics. The functions include at the WORLD layer:

i) Device cached,

ii) Device un-cached,

iii) Device moves from one ambient to another (This can be modelled by application of ‘device un-cached’ rule in the initial ambient and ‘device cached’ rule in the final ambient.),

And at the Service Component Architecture Layer (SCA),

iv) State of a service changes.

Finally, to modify the Bigraphical array `compositionDevices` we have written the following two functions to encapsulate the appropriate reaction rules:

v) A function that encapsulates the rule that a device joins `compositionDevices` .

vi) A function that encapsulates the rule that a device needs to be removed from `compositionDevices` .

The reaction rule encapsulated by a function along with the Bigraph which needs to be rewritten, are passed on to the matching algorithm which returns with an appropriate match. The matching algorithm is encapsulated in the function `changeSystem` which takes in a Bigraph `system` and a reaction rule `ruleName` as parameters and returns the modified system.

We now discuss the function `changeSystem` in detail (Figure 5-6).

In line 3, we first convert the Bigraph that has been passed to `changeSystem` from a BGVal to BDnf using Elsborg's code (Elsborg, 2009). Next, in line 4 and 5, we compute a lazy list of the matches of redex with the Bigraph that needs to be rewritten using Elsborg's code (Elsborg, 2009). We then access the first element of the lazy list using Elsborg's code (Elsborg, 2009) in line 6. Notice that we expect only one match because all device ids, location ids, and service ids that our functions use to construct the reaction rules are unique. Finally, in line 8, we perform a single reaction step induced by a match (Elsborg, 2009) returning the resulting re-written Bigraph. The full function has been written by us by collecting together lines from Elsborg's code (Elsborg, 2009) . See Figure 5-6.

```
1  fun changeSystem(systemName,ruleName) =
2   let
3    val BRsystemName = makeBR systemName
4    val mtsd = M.matches { agent = BRsystemName ,
5    rule = ruleName }
6    val matchNew = LazyList.lznth mtsd 0
7   in
8    Re.react matchNew
9   end
```

FIGURE 5-6: FUNCTION `changeSystem`.

5.3.1.1 ABSTRACTION BY PARAMETERIZATION

Abstraction by parameterization *'through the introduction of parameters, allows us to represent a potentially infinite set of different computations with a single program text that is an abstraction of all of them*'(Liskov and Guttag, 2000). We look upon our functions as generic templates from which reaction rules can be automatically instantiated. Our functions that encapsulate reaction rules parameterize the identities of locations, devices and services for a given reaction rule as the case may be. Note that the reaction rules as defined in Bigraph theory are themselves parametric because they have sites that can be 'filled up' (See Chapter 2). The parameters that our functions provide to the reaction rules are in addition to the parameters that 'fill up' the sites of those reaction rules. We now discus the following six functions that modify the structure of the WORLD layer: i) device un-cached, ii) device cached, iii) device moves from one ambient to another, a fourth function that modifies the state of the SCA layer: iv) state of a service changes, v) a function that encapsulates the rule that a device joins `compositionDevices` , and finally, vi) a function that encapsulates the rule that a device needs to be removed from `compositionDevices` .

Each of these functions can automatically instantiate infinitely many reaction rules.

i) Device un-cached rule: We use the 'device disappears' rule to un-cache a device from the WORLD layer. Consider, again, the application of the reaction rule to un-cache a device (see Figure 5-1) to the state of the WORLD shown in Figure 5-2. This reaction rule models the un-caching of device `i4.3` from location `i1`. The signature of the function which encapsulates this rule is `constructDisappear (deviceId, location)` and the structure of the resulting reaction rule is as shown in Figure 5-7. The parameters `deviceId` and `location` are the parameters passed by the function to the parametric reaction rule. The function returns a constructed reaction rule. When we pass `deviceId = i4.3` and `location = i1`, to the reaction rule of Figure 5-7, we get the reaction rule of Figure 5-1. Compared to Figure 5-1, the function `constructDisappear(deviceId,location)` that encapsulates the reaction rule of Figure 5-7 can therefore instantiate an infinite number of reaction rules for an infinite number of `deviceIds` and `locations`. As discussed earlier in this section, the site in the redex and reactum of the reaction rule is a don't care. It is the reaction rule's parameter in which a Bigraph could be fitted.

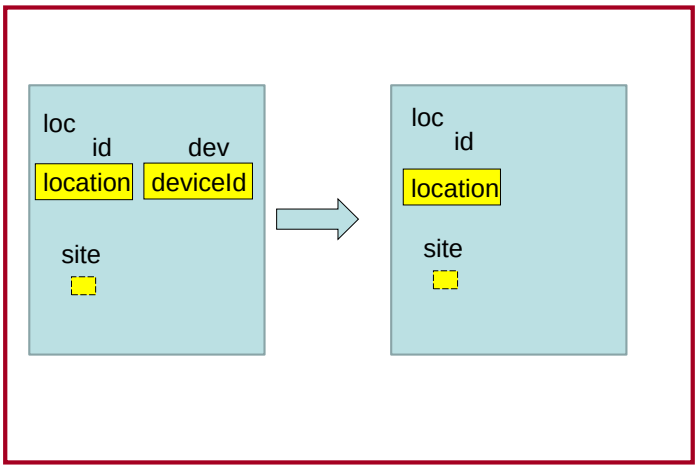

FIGURE 5-7: FUNCTION `constructDisappear`.

We now discuss `constructDisappear(deviceId,location)` function's implementation in MiniML. The input parameters for the function are the identity of the device (`deviceId`) and the location of the device (`location`). The function returns the reaction rule constructed for those parameters. This function can generate infinitely many reaction rules -depending on the input parameters- with the site of the reaction rule playing the role of 'don't care' -in the sense that it does not matter which Bigraph is substituted in place of the site for a reaction rule to be applied.

We now discuss the function in detail (See Figure 5-8). Firstly, in lines 3 to 5, we declare the structure of the redex and reactum. Notice that we use functions `loc''` (discussed in section 4.4.1.1) to construct location with the id `location`. Similarly, we use the function `constructDevice` (discussed in section 4.4.1.1.1) to construct the device with the id `deviceId`. Next, in lines 6 and 7, we specify that the number of the sites in redex and reactum is one each. In lines 8 to 10, we map the site of the reactum to the site of the redex. Finally, in lines 12 to 16, we use the 'make' function to construct the reaction rule. Here, structure R = BG.Rule , where 'Rule' is the abstract data type provided by BPL Tool for constructing rules. Notice that the parameters `deviceId` and `location` are unique in our model. Thus, the redex that gets passed to the matching algorithm always has a unique device id and location id and we get a single match for the redex in the large Bigraph where a match is being searched for.

```
fun constructDisappear(deviceId,location)=
 let
  val redexDisappear = S.o
  (loc''(location),S.`|`(constructDevice(deviceId),site))
  val reactDisappear = S.o (loc''(location),site)
  val redex_innerface_Disappear = Iface.m 1
  val react_innerface_Disappear = Iface.m 1
  val instDisappear = Inst.make { I = redex_innerface_Disappear,
  J = react_innerface_Disappear,
  maps = [((0,[]), (0,[]))] }
 in
  R.make { name = "Disappear",
  redex = makeBR redexDisappear,
  react = reactDisappear,
  inst = instDisappear,
  info = info }
 end
```

FIGURE 5-8: FUNCTION `constructDisappear`.

We have written another function `deviceDisappears` that uses the reaction rule returned by the function `constructDisappear` to change the state of the system. Function `deviceDisappears` takes the id of device (`deviceId`), the location of device (`location`), and the current state of the system (`system`) to return the new state of the system. This function is shown in Figure 5-9.

```
fun deviceDisappears(deviceId,location,system)=
 let
  val Disappear = constructDisappear(deviceId,location)
 in
  changeSystem(system,Disappear)
 end
```

FIGURE 5-9: FUNCTION `deviceDisappears`.

ii) Device Cached rule: This rule shown in Figure 5-10 is encapsulated by a function `constructAppear(deviceId,location)`.

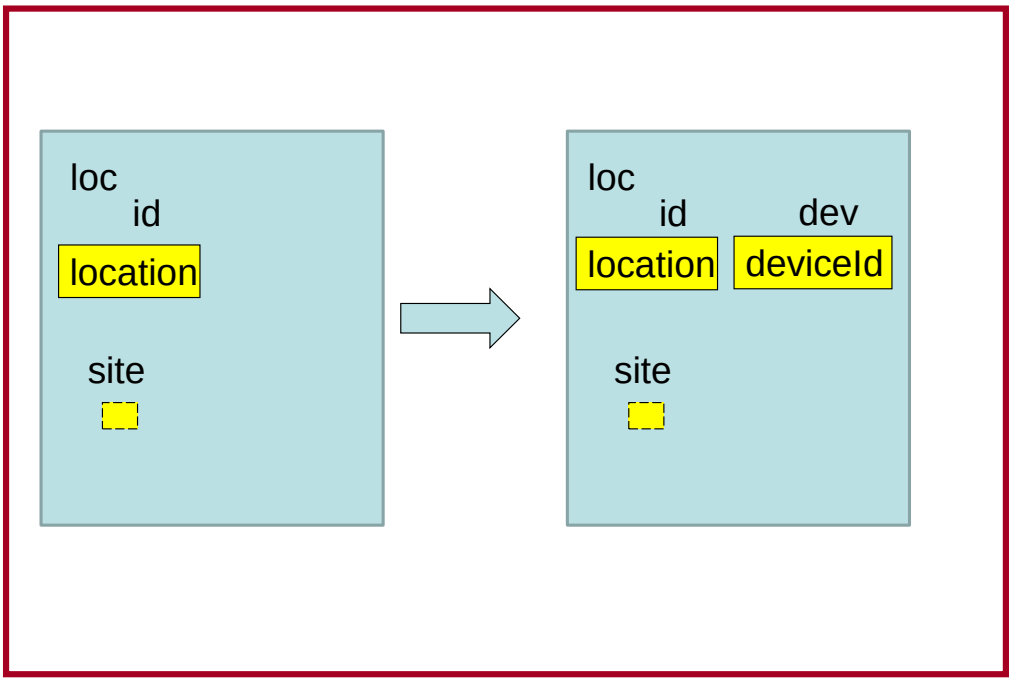

FIGURE 5-10: DEVICE CACHED REACTION RULE.

The input parameters are: the id of the device (`deviceId`) and the id of the location (`location`). The function returns the reaction rule constructed for those parameters. This function can generate infinitely many reaction rules -depending on the input parameters- with the site of the reaction rule playing the role of 'don't care' -in the sense that it does not matter which Bigraph is in the same place as the site, for a reaction rule to be applied.

We now discuss the function in detail (See Figure 5-11). Firstly, in lines 3 to 5 we declare the structure of the redex and reactum. Notice that we use functions `loc''`(discussed in section 4.4.1.1) to construct location with the id `location`. Similarly, we use the function `constructDevice` (discussed in section 4.4.1.1.1) to construct the device with the id `deviceId`. Next, in lines 6 and 7 we specify that the number of the sites in redex and reactum is one each. In lines 8 to 10, we map the site of the reactum to the site of the redex. Finally, in lines 12 to 16, we use the 'make' function to construct the reaction rule. Here, structure R = BG.Rule , where 'Rule' is the abstract data type provided by BPL Tool for constructing rules. Notice that the parameters `deviceId` and `location` are unique in our model. Thus, the redex that gets passed to the matching algorithm always has a unique device id and location id and we get a single match for the redex in the Bigraph where a match is being searched for.

```
fun constructAppear(deviceId,location)=
 let
  val redexAppear = S.o (loc''(location),site)
  val reactAppear = S.o
  (loc''(location),S.`|`(constructDevice(deviceId),site))
  val redex_innerface_Appear = Iface.m 1
  val react_innerface_Appear = Iface.m 1
  val instAppear = Inst.make { I = redex_innerface_Appear,
  J = react_innerface_Appear,
  maps = [((0,[]), (0,[]))] }
 in
  R.make { name = "Appear",
  redex = makeBR redexAppear,
  react = reactAppear,
  inst = instAppear,
  info = info }
 end
```

FIGURE 5-11: FUNCTION `constructAppear`.

We have written another function `deviceAppears` that uses the reaction rule returned by the function `constructAppear` to change the state of the system. Function `deviceAppears` takes the id of device (`deviceId`), the location of device (`location`), and the current state of the two layer system (`system`) to return the new state of the system. This function is shown in Figure 5-12.

```
fun deviceAppears(deviceId,location,system)=
 let
  val Appear = constructAppear(deviceId,location)
 in
  changeSystem(system,Appear)
 end
```

FIGURE 5-12: FUNCTION `deviceAppears`.

iii) 'Device moves from one ambient to another' rule: We use this rule to model in the WORLD layer, the movement of the user's device. As already discussed, we designate the user's device as `devi0`. This movement can be modelled by application of 'device un-cached' rule (encapsulated within `deviceDisappears` function) in the initial ambient and 'device cached' rule (encapsulated within `deviceAppears` function) in the final ambient. The input parameters for the function are the id of the device (`deviceId`), initial location of the device (`initialLocation`), final location of device (`finalLocation`) and the current state of the

Bigraph (`system`). The function returns a modified Bigraph with the device in the final location as shown in Figure 5-13.

```
1  fun changeAmbient(deviceId,initialLocation,finalLocation,system)=
2   let
3    val systema =
4    deviceDisappears(deviceId,initialLocation,system)
5   in
6    deviceAppears(deviceId,finalLocation,systema)
7   end
```

FIGURE 5-13: FUNCTION `changeAmbient`.

iv) As discussed earlier (Sections 4.4.2.5), each reaction rule of the SCA layer models a change in state. The possible states are: 1) Unresponsive, 2) Incorrect result, 3) Incoherent results, 4) Slow service 5) Outdated results. 6) Working, 7) Not working (models any other fault that we have not captured). These states are represented as nodes in our model.

We now discuss the function `constructStateChange` that encapsulates a reaction rule that changes state of a service at the SCA layer. The input parameters are the id of the service (`serviceId`), initial state of the service (`initialState`) and the final state of the service (`finalState`). The function returns the reaction rule constructed for those parameters. Again, we can use this function to generate infinitely many reaction rules. The reaction rule is shown in Figure 5-14.

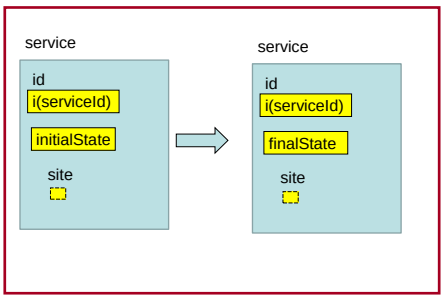

FIGURE 5-14: `constructStateChange` FUNCTION ENCAPSULATING THE REACTION RULE THAT CHANGES STATE OF A SERVICE.

We now discuss the code for this function (See Figure 5-15). Firstly, in lines 3 and 4, we declare the structure of the redex and reactum. We use the function `constructService` (discussed in section 4.4.2.1) to construct the redex and reactum of the reaction rule. Redex represents a service in the initial state and reactum represents a service in the final state. Next, in lines 5 and

6, we specify that the number of the sites in redex and reactum is one each. In lines 7 and 8, we map the site of the reactum to the site of the redex. Finally, in lines 10 to 14, we use the 'make' function to construct the reaction rule. Here, structure R = BG.Rule , where 'Rule' is the abstract data type provided by BPL Tool for constructing rules. Notice that the parameter `serviceId` is unique in our model. Thus, the redex that gets passed to the matching algorithm always has a unique service id and we get a single match for the redex in the large Bigraph where a match is being searched for.

```
1  fun constructStateChange(serviceId,initialState,finalState) =
2   let
3    val redexState = constructService(serviceId,initialState)
4    val reactState = constructService(serviceId,finalState)
5    val redex_innerface_State = Iface.m 1
6    val react_innerface_State = Iface.m 1
7    val instState = Inst.make { I = redex_innerface_State,
8    J = react_innerface_State, maps=[((0,[]), (0,[]))] }
9   in
10   R.make { name = "stateChange",
11   redex = makeBR redexState,
12   react = reactState,
13   inst = instState,
14   info = info }
15  end
```

FIGURE 5-15: FUNCTION `constructStateChange`.

v) Device joins Bigraphical array rule: This rule is shown in Figure 5-16. When a device's service joins the service composition, that device needs to be added to the Bigraphical array `compositionDevices`. We have written a function

`constructDeviceJoinsCompositionRule`

encapsulating the reaction rule that adds a device to `compositionDevices`.

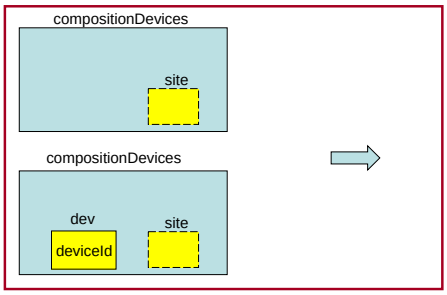

FIGURE 5-16: DEVICE JOINS COMPOSITION RULE

The input parameter is the id of the device (`deviceId`). The function returns the reaction rule constructed for this parameter. Once again, notice that like other functions encapsulating reaction rules, this function too can generate infinitely many reaction rules -depending on the input parameters. Also, the site of the reaction rule plays the role of 'don't care'. That is to say that it does not matter which Bigraph is in the same place as the site, for a reaction rule to be applied.

We now discuss the function in detail (See Figure 5-17). Firstly, in lines 3 to 5 we declare the structure of the redex and reactum. Notice that we use the function `constructDevice` (discussed in section 4.4.1.1.1) to construct the device with the id `deviceId`. Next, in lines 6 and 7 we specify that the number of the sites in redex and reactum is one each. In lines 8 to 10, we map the site of the reactum to the site of the redex. Finally, in lines 12 to 16, we use the 'make' function to construct the reaction rule. Here, structure R = BG.Rule , where 'Rule' is the abstract data type provided by BPL Tool for constructing rules.

```
1  fun constructDeviceJoinsCompositionRule (deviceId) =
2   let
3    val redexAppear = S.o (compositionDevices,site)
4    val reactAppear = S.o
5    (compositionDevices,S.`|`(constructDevice(deviceId),site))
6    val redex_innerface_Appear = Iface.m 1
7    val react_innerface_Appear = Iface.m 1
8    val instAppear = Inst.make { I = redex_innerface_Appear,
9    J = react_innerface_Appear,
10   maps = [((0,[]), (0,[]))] }
11  in
12   R.make { name = "deviceAdded",
13   redex = makeBR redexAppear,
14   react = reactAppear,
15   inst = instAppear,
16   info = info }
17  end
```

FIGURE 5-17: FUNCTION `constructDeviceJoinsCompositionRule`

We have written another function `deviceJoinsComposition` that uses the reaction rule returned by the function `constructDeviceJoinsCompositionRule` to change the state of the system. Function `deviceJoinsComposition` takes the id of device (`deviceId`) and the current state of the two layer system (`system`) to return the new state of the system. This function is shown in Figure 5-18.

```
1  fun deviceJoinsComposition(deviceId, system) =
2   let
3    val deviceAdded = constructDeviceJoinsCompositionRule(deviceId)
4   in
5    changeSystem(system,deviceAdded)
6   end
```

FIGURE 5-18: FUNCTION `deviceJoinsComposition`.

vi) Device is removed from the Bigraph array rule: This rule is shown in Figure 5-19. When a device's service develops a fault, that device needs to be removed from the Bigraphical array `compositionDevices`. We have written a function

`constructDeviceLeavesCompositionRule`

encapsulating the reaction rule that removes a device from `compositionDevices`.

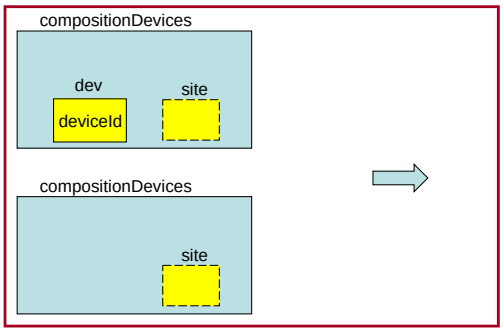

FIGURE 5-19: DEVICE LEAVES COMPOSITION RULE

The input parameter is the id of the device (`deviceId`). The function returns the reaction rule constructed for this parameter. Once again, notice that like other functions encapsulating reaction rules, this function too can generate infinitely many reaction rules -depending on the input parameters. Also, the site of the reaction rule plays the role of 'don't care'. That is to say that it does not matter which Bigraph is in the same place as the site, for a reaction rule to be applied.

We now discuss the function in detail (See Figure 5-20). Firstly, in lines 3 to 5 we declare the structure of the redex and reactum. Notice that we use the function `constructDevice` (discussed in section 4.4.1.1.1) to construct the device with the id `deviceId`. Next, in lines 6 and 7 we specify that the number of the sites in redex and reactum is one each. In lines 8 to 10, we map the site of the reactum to the site of the redex. Finally, in lines 12 to 16, we use the 'make' function to construct the reaction rule. Here, structure R = BG.Rule , where 'Rule' is the

abstract data type provided by BPL Tool for constructing rules. Notice that the parameters `deviceId` is unique in our model. Thus, the redex that gets passed to the matching algorithm always has a unique device id and we get a single match for the redex in the Bigraph where a match is being searched for.

```
1  fun constructDeviceLeavesCompositionRule (deviceId) =
2   let
3    val redexDisappear = S.o
4    (compositionDevices,S.`|`(constructDevice(deviceId),site))
5    val reactDisappear = S.o (compositionDevices,site)
6    val redex_innerface_Disappear = Iface.m 1
7    val react_innerface_Disappear = Iface.m 1
8    val instDisappear= Inst.make { I = redex_innerface_Disappear,
9    J = react_innerface_Disappear,
10   maps = [((0,[]), (0,[]))] }
11  in
12   R.make { name = "deviceDeleted",
13   redex = makeBR redexDisappear,
14   react = reactDisappear,
15   inst = instDisappear,
16   info = info }
17  end
```

FIGURE 5-20: FUNCTION `constructDeviceLeavesCompositionRule`

We have written another function `deviceLeavesComposition` that uses the reaction rule returned by the function `constructDeviceLeavesCompositionRule` to change the state of the system. Function `deviceLeavesComposition` takes the id of device (`deviceId`) and the current state of the two layer system (`system`) to return the new state of the system. This function is shown in Figure 5-21.

```
1  fun deviceLeavesComposition(deviceId, system) =
2   let
3    val deviceDeleted = constructDeviceLeavesCompositionRule(deviceId)
4   in
5    changeSystem(system, deviceDeleted)
6   end
```

FIGURE 5-21: FUNCTION `deviceLeavesComposition`.

Altogether, we see that the implementation of functions that modify PGM-like models depend on the matching algorithm and parameterization concepts are needed to write functions that generate infinitely many reaction rules.

5.3.2 FUNCTIONS THAT ACCESS INFORMATION FROM THE MODEL

We want to 'interrogate' the WORLD/SCA model to access information that is needed to take an adaptation decision at runtime. The BPL Tool provides no functions to access the structure of a Bigraph. One of the ways this can be overcome is by writing reaction rules with the same redex and reactum and passing the redex to the matching algorithm. The redex is written in such a fashion that the parameters returned by the matching algorithm can be used to extract useful information using BPL Tool functions. Notice, because reactum is the same as redex, it does not make sense to replace redex by reactum in the Bigraph where matching occurs. Rather, we are interested in the parameters returned by the matching algorithm. Hence we do not apply the matches returned by the matching algorithm to the Bigraph on which the reaction rule was originally applied. We have written the following functions using this strategy, and we now discuss this strategy in more detail for each of these function:

i) `locateDevice:` Finds where a device is located

ii) `enumerateDevicesInShoppingMall:` Lists all devices in a particular ambient (location) offering a particular service.

iii) `findParent:` Finds a location's parent location.

iv) `findChild`: Finds one of the child locations of a location.

v) `newFindParticipatingDevice`: Finds the id of the device participating in the composition and offering a particular service.

vi) `constructSeviceTree`: Returns a list of all the services participating in the composition by using the BPL Tool's matching algorithm to traverse the tree representing the structure of the service composition at the SCA layer.

We now discuss each of these functions in turn:

i) `locateDevice`: We use this function to find where a device is located by accessing the information stored in the WORLD Bigraph through the matching algorithm. Consider the state of the WORLD layer in Figure 5-2 which we reproduce in Figure 5-22. Suppose that we wish to find the location of device `i4.3`.

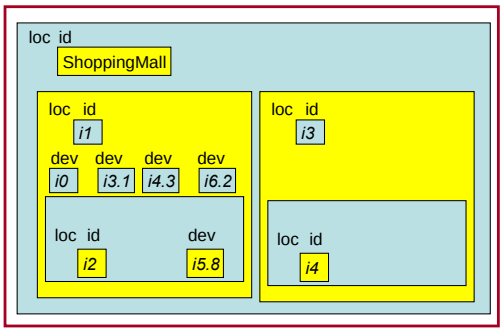

FIGURE 5-22: INITIAL STATE OF WORLD BIGRAPH.

We use the reaction rule shown in Figure 5-23. Notice from the figure that both the redex and reactum are the same Bigraphs. We pass the redex with `deviceId = 4.3` together with the WORLD layer Bigraph of Figure 5-22 to the matching algorithm. Notice that we do not specify the location Id of the device in the reaction rule because it is an unknown and we want the matching algorithm to return the parameter containing that information.

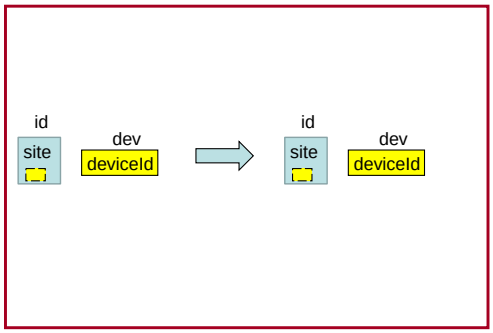

FIGURE 5-23: REACTION RULE WITH THE SAME REDEX AND REACTUM.

The context and the parameter that the matching algorithm returns are shown in Figures 5-24 and 5-25 respectively. Notice that the 'don't care' (site) of the reaction rule in Figure 5-23 is 'filled up' by the parameter in the Figure 5-25. Thus, the BPL Tool's matching algorithm returns a parameter that is the atomic Bigraphical node representing the id of the location: `i1`.

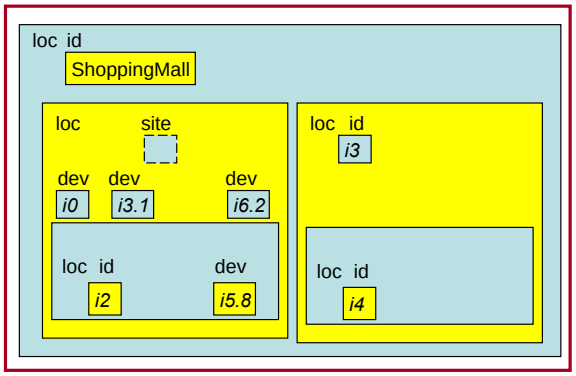

FIGURE 5-24: CONTEXT RETURNED BY THE MATCHING ALGORITHM.

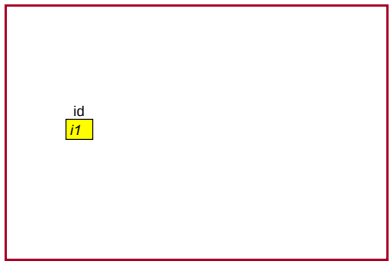

FIGURE 5-25: PARAMETER RETURNED BY THE MATCHING ALGORITHM.

We now discuss the code for function `locateDevice` shown in Figure 5-26. The input parameters for the function are the id of the device (`deviceId`) and the current state of the two layer model (`currentSystem`). The function returns the string representation of the id of location.

Firstly, we declare the structure of the redex in line 3 and reactum in line 4. Notice that both of them represent the same Bigraph. Notice too that there is no identity node for location node. We want the matching algorithm to return the string representation of identity node as part of the parameter that it will return. Next, in lines 5 and 6 respectively, we specify that the number of the sites in redex and reactum is one each. We map the site of the reactum to the site of the redex in lines 7 to 9. We use the `make` function to construct the reaction rule in lines 10 to 14. Here, structure R = BG.Rule , where 'Rule' is the abstract data type provided by BPL Tool for constructing rules. In line 15, we then convert the Bigraph that has been passed to `locateDevice` function from a BGVal to BDnf using Elsborg's code (Elsborg, 2009). Next, we compute a lazy list of the matches of redex with the Bigraph that needs to be rewritten using Elsborg's code (Elsborg, 2009) in lines 16 to 17. Notice that we expect only one match because the parameter `deviceId` passed to `locateDevice` is unique.

In line 18, we check whether the lazy list containing the matches is empty. When the lazy list is not empty, the lines 19 to 32 execute. In lines 21 to 25, we use BPL Tool's functions to extract out the Bigraph representing the parameter returned by the matching algorithm. Next, in line 26, we define the function `peel` adapted from Elsborg's code (Elsborg, 2009) to convert the BGVal representation of a Bigraph to a string representation. Finally, we return this sting representation of the id of the location in line 34.

```
1  fun locateDevice(deviceId,currentSystem) =
2   let
3    val redexLocate = S.`|`(S.o(id,site),constructDevice(deviceId))
4    val reactLocate = S.`|`(S.o(id,site),constructDevice(deviceId))
5    val redex_innerface_Locate = Iface.m 1
6    val react_innerface_Locate = Iface.m 1
7    val instLocate = Inst.make { I = redex_innerface_Locate,J =
8    react_innerface_Locate,
9    maps = [((0,[]), (0,[]))] }
10   val findDevice =  R.make { name = "findDevice",
11   redex = makeBR redexLocate,
12   react = reactLocate,
13   inst = instLocate,
14   info = info }
15   val BRsystemLocate = makeBR currentSystem
16   val mtsd = M.matches { agent = BRsystemLocate , rule = findDevice
17   }
18   val testNew = if ((lzLength mtsd) = "0") then ""
19                 else
20                  let
21                   val testalteragent' = M.unmk (LazyList.lzhd mtsd)
22                   val testalteragent'_par =
23                   #parameter(testalteragent')
24                   val testalteragent'_ctx =
25                   #context(testalteragent')
26                   fun peel x = (B.toString o B.simplify o
27                                 Bdnf.unmk) x
28                   val test3 = peel testalteragent'_par
29                   val test4 = peel testalteragent'_ctx
30                  in
31                   test3
32                  end
33  in
34   testNew
35  end
```

FIGURE 5-26: FUNCTION `locateDevice`.

To sum up, the function `locateDevice` uses a reaction rule that does not change the state of the WORLD layer to access information. This is achieved by having the same redex and reactum for the reaction rule. This reaction rule is then passed on to the matching algorithm and it returns parameters from which the location can be extracted.

ii) `enumerateDevicesInShoppingMall:` This function lists all devices in a particular ambient (location) offering a particular service by accessing the Bigraph through the use of the matching algorithm. The function is used when a device needs to be replaced and as a result, we want to find out which other device is offering that same service in a particular ambient. Once we know an alternative device offering the same service, we can send out a command to bind our service composition to the new service.

Consider the state of the WORLD layer in Figure 5-27. Suppose that we want to find all the devices in location `1` that offer service `6`. There are two such devices: devices with id `i6.2` and `i6.4`.

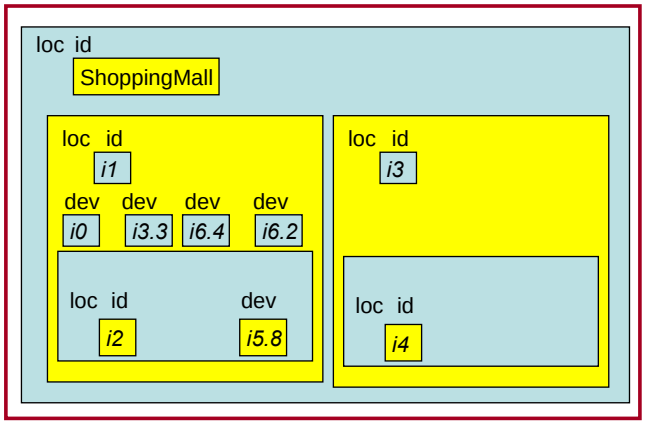

FIGURE 5-27: INITIAL STATE OF WORLD BIGRAPH.

We use the reaction rule shown in Figure 5-28. Notice that both redex and reactum are the same Bigraphs. We pass redex with `location = 1` and `serviceId = 6` as well as the Bigraph with the state of the WORLD layer shown in Figure 5-27 to the matching algorithm.

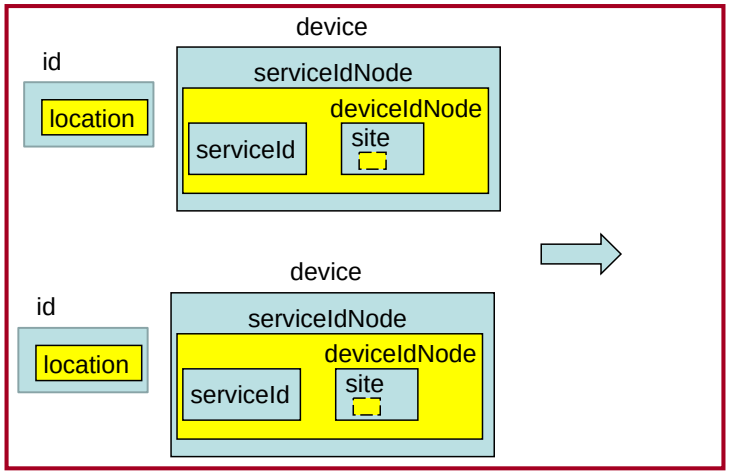

FIGURE 5-28: REACTION RULE WITH THE SAME REDEX AND REACTUM.

Notice that for the reaction rule shown in Figure 5-28, the matching algorithm returns two sets of contexts and parameters. This is because there are two matches corresponding to two values of ordinal numbers for `deviceIdNodes`: `4` and `2` for the single value of `serviceId = 6`.

The first set of context and the parameters that the matching algorithm returns are shown in Figures 5-29 and 5-30 respectively. Notice that the 'Don't care' (site) of the reaction rule in Figure 5-28 is filled up by the parameter in Figure 5-30.

Similarly, the second set of context and the parameters that the matching algorithm returns are shown in Figures 5-31 and 5-32 respectively. Once again, notice that the 'Don't care' (site) of the reaction rule in Figure 5-28 is filled up by the parameter in Figure 5-32.

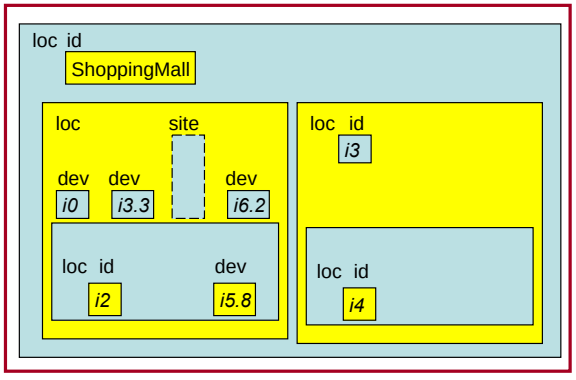

FIGURE 5-29: FIRST OF THE TWO CONTEXTs RETURNED BY MATCHING ALGORITHM-ONLY `i6.4` ABSENT.

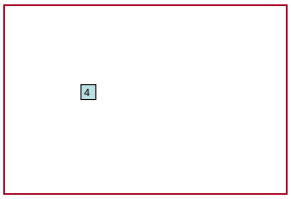

FIGURE 5-30: FIRST OF THE TWO PARAMETERs RETURNED BY THE MATCHING ALGORITHM ONLY 4 PRESENT.

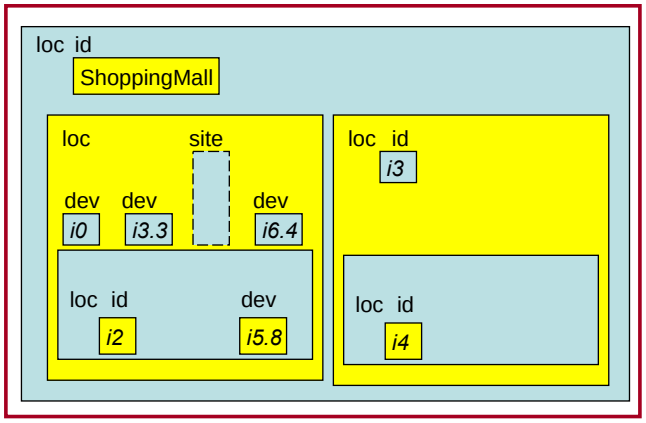

FIGURE 5-31: SECOND OF THE TWO CONTEXTS RETURNED BY MATCHING ALGORITHM- ONLY i6.2 ABSENT.

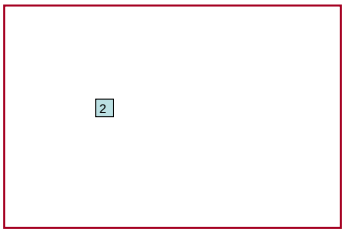

FIGURE 5-32: SECOND OF THE TWO PARAMETERS RETURNED BY THE MATCHING ALGORITHM- ONLY 2 PRESENT.

We now discuss the code of the function in detail. The code is shown in Figure 5-33. The input parameters for the function are the string representation of the id of the

service (seviceId), the string representation of id of location (location), and the BGVal representation of the current state of WORLD and SCA layers (currentSystem). The function returns a list of all devices in a particular ambient offering a particular service.

Firstly, we declare the structure of the redex in line 3 and reactum in line 7. Notice that both of them represent the same Bigraph. We pass the identity of location (location) and the service Id (serviceId) to the redex and reactum. Next, in lines 11 and 12 respectively, we specify that the number of the sites in redex and reactum is one each. Then, we map the site of the reactum to the site of the redex in lines 14 to 17. We next use the make function to construct the reaction rule in lines 19 to 24. Here, structure R = Bg.Rule where 'Rule' is the abstract data type provided by BPL tool for constructing rules. In line 26, we convert the Bigraph that has been passed to enumerateDevicesInShoppingMall function from a BGVal to BDnf using Elsborg's code (Elsborg, 2009) so that we can pass it on to the matching algorithm. Next, we compute a lazy list of the matches of redex with the Bigraph that needs to be rewritten using Elsborg's code for the matching algorithm in lines 27 and 28. Notice that we expect more than one match because the parameter serviceId passed to enumerateDevicesInShoppingMall could be associated with more than one device in a given location. We then use Elsborg's code to define a curried function locationIdentityStringList in lines 30 to 38. It takes in a BDnf representation of a bigraph and a Match type (of BPL Tool), as input parameters. This function locationIdentityStringList then extracts the parameter in the BDnf form from the match in line 33, and converts it in line 35 into a string using the peel function of the BPL Tool. In line 34, we define the function peel adapted from Elsborg's code to convert the BGVal representation of a Bigraph to a string representation. On line 40, we pass the function locationIdentityStringList with its BDnf parameter BRsystemLocateInShoppingMall and the mtsd lazy list to the curried function LazyList.lzmap provided by the BPL Tool. The function LazyList.lzmap applies our function locationIdentityStringList to each element of the lazy list mtsd and returns a lazy list consisting of the string representations of the BDnf representation of the ordinal numbers of the devices. Then in line 43, we convert the lazy list locIdStringList into a list. In line 44, we define the function prefixServiceId that prefixes a string representation of the serviceId to the string representation of an ordinal number. Next, in line 45, we define a function zeroServiceId that converts its parameter into the string "0". Finally, we use both of these functions from lines 47 to 52 to return a string list of correctly constructed device Ids (with service Ids and ordinal numbers).

```
fun enumerateDevicesInShoppingMall(serviceId,location,currentSystem)=
 let
  val redexLocateInShoppingMall = S.`|` (S.o (id, i(location)),
  S.o(device,S.o(serviceIdNode,S.`|`
  (i(serviceId),S.o(devIdNode,site)))))

  val reactLocateInShoppingMall = S.`|` (S.o (id, i(location)),
  S.o(device,S.o(serviceIdNode,S.`|`
  (i(serviceId),S.o(devIdNode,site)))))

  val redex_innerface_LocateInShoppingMall = Iface.m 1
  val react_innerface_LocateInShoppingMall = Iface.m 1

  val instLocateInShoppingMall = Inst.make { I =
  redex_innerface_LocateInShoppingMall,
  J = react_innerface_LocateInShoppingMall,
  maps = [((0,[]), (0,[]))] }

  val findDeviceInShoppingMall = R.make { name =
  "findDeviceInShoppingMall",
  redex = makeBR redexLocateInShoppingMall,
  react = reactLocateInShoppingMall,
  inst = instLocateInShoppingMall,
  info = info }

  val BRsystemLocateInShoppingMall = makeBR currentSystem
  val mtsd = M.matches { agent = BRsystemLocateInShoppingMall ,
  rule = findDeviceInShoppingMall }

 fun locationIdentityStringList agent m =
  let
   val agent' = M.unmk (m)
   val agent'_par = #parameter(agent')
   fun peel x = (B.toString o B.simplify o Bdnf.unmk) x
   val test1 = peel agent'_par
  in
   test1
  end

 val locIdStringList = LazyList.lzmap (locationIdentityStringList
 BRsystemLocateInShoppingMall) mtsd

 val aList = LazyList.lztolist locIdStringList
 fun prefixServiceId aserviceId x = aserviceId ^ "." ^ x
```

```
45   fun zeroServiceId y = "0"
46
47   val correctServiceIdList = if(serviceId = "0")then
48   map(zeroServiceId)
49   else map(prefixServiceId serviceId)
50
51  in
52   correctServiceIdList aList
53  end
```

FIGURE 5-33: FUNCTION `enumerateDevicesInShoppingMall`.

In conclusion, in the function `enumerateDevicesInShoppingMall`, we pass a reaction rule with the same redex and reactum to the matching algorithm so that we can access the device Ids offering a backup of a particular service in a particular location. Notice that the state of the WORLD does not change on application of the reaction rule. This is as it should be, because, we only want to access information from the WORLD Bigraph, and, not modify it.

iii) `findParent`: We use this function to find a location's parent location. Consider again the state of the WORLD in Figure 5-27 which we reproduce in Figure 5-34. Suppose, we wish to find the parent of location 2.

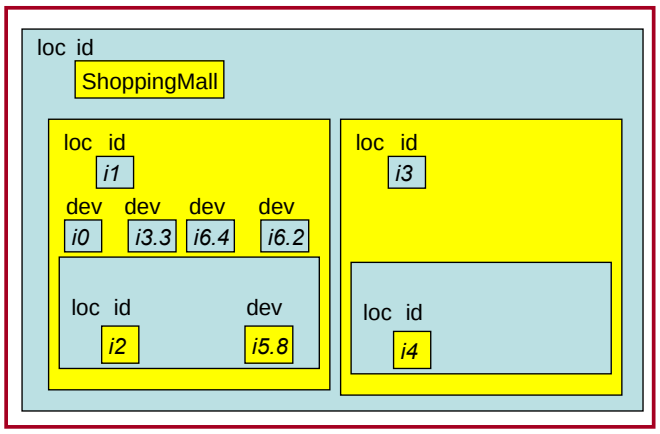

FIGURE 5-34: INITIAL STATE OF WORLD BIGRAPH.

We use the reaction rule shown in Figure 5-35. Notice from the Figure that both the redex and reactum are the same Bigraphs. We pass the redex with `location` = "2" together with the WORLD layer Bigraph of Figure 5-34 to the matching algorithm.

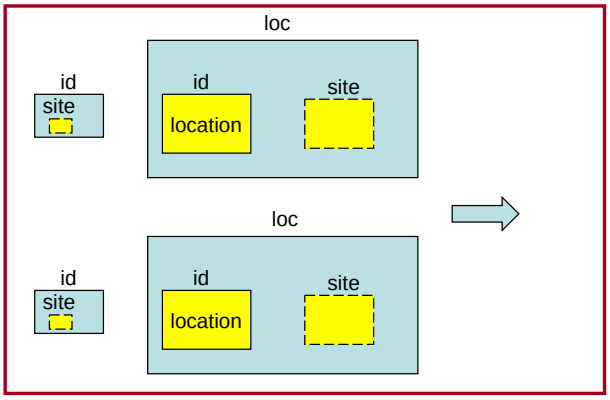

FIGURE 5-35: REACTION RULE WITH THE SAME REDEX AND REACTUM.

The context and the parameter that the matching algorithm returns are shown in Figures 5-36 and 5-37 respectively. Notice that the 'don't care' (site) of the reaction rule in Figure 5-35 is 'filled up' by the parameter in the Figure 5-37. Two Bigraphical parameters corresponding to the two sites in the redex of the reaction rule are returned. In Figure 5-37, the two Bigraphical parameters that have been returned are shown within dashed boxes. These dashed boxes represent the root outer interfaces of the Bigraphs as discussed in chapter 2. We can now extract out the first Bigraph from these two parameters.

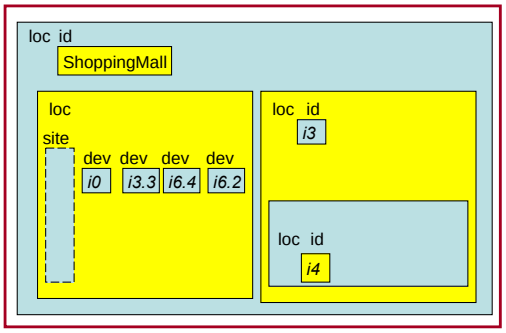

FIGURE 5-36: CONTEXT RETURNED BY MATCHING ALGORITHM.

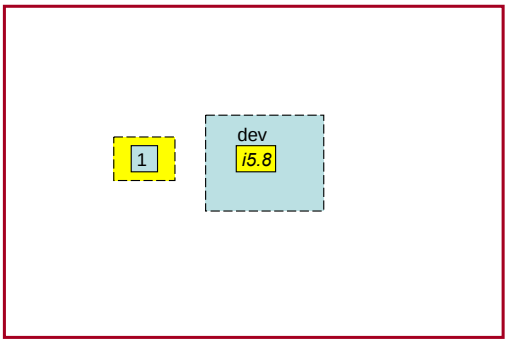

FIGURE 5-37: PARAMETER RETURNED BY MATCHING ALGORITHM.

We now discuss the code of this function shown in Figure 5-38.

The input parameters for the function are the id of the location whose parent we wish to find (`location`) and the current state of the two layer model (`currentSystem`). The function returns the string representation of the id of the parent location.

Firstly, on line 2 we check if the value of `location` is `ShoppingMall`. If that is the case, we return an empty string as `ShoppingMall` location has no parent. Next, from lines 3 to 34, we define the else statement. We declare the structure of the redex in line 5 and reactum in line 6. Next, in lines 7 and 8 respectively, we specify that the number of the sites in redex and reactum is two each (See Figure 5-35). We map the sites of the reactum to the sites of the redex in lines 9 to 12. We use the `make` function to construct the reaction rule in lines 13 to 17. Here, structure R = BG.Rule , where 'Rule' is the abstract data type provided by BPL Tool for constructing rules. In line 18, we then convert the Bigraph that has been passed to `findParent` function from a BGVal to BDnf using Elsborg's code (Elsborg, 2009). Next, we compute a lazy list of the matches of redex with the Bigraph that needs to be rewritten using Elsborg's code (Elsborg, 2009) in line 19. Notice that we expect only one match because the parameter `location` passed to `findParent` is unique. From lines 20 to 31, we use an if-else expression to construct the parent location's identity string. In line 20, we write a guard condition and return an empty string if the length of the list containing the matches is zero. Line 21 starts the else branch of the expression. Since there is only one match, we extract that match in line 23 as the head of the lazy list. Next, in line 24, we extract only the parameters from the match (since a match contains both the context and the parameters and we are interested only in the parameter). Then, in line 25, we deconstruct the parameter (which is of BDnf type) into its constituent link and place Bigraphs. Since we do not use links in our modeling, the constituent link returned will be empty and we will only have a list of place Bigraphs. In line 26, we extract the list of place graphs from the deconstructed parameter. Finally, in line 27 we access the first Bigraph in the list as it

will be the parent location's id. We then convert this Bigraph into its string form in line 28 and return this string in line 33.

```
1  fun findParent(location,currentSystem) =
2   if(location = "ShoppingMall") then ""
3   else
4    let
5     val redexParent = S.`|`(S.o(id,site),loc''(location))
6     val reactParent = S.`|`(S.o(id,site),loc''(location))
7     val redex_innerface_Parent = Iface.m 2
8     val react_innerface_Parent = Iface.m 2
9     val instParent = Inst.make { I = redex_innerface_Parent,
10    J = react_innerface_Parent,
11    maps = [((0,[]), (0,[])),
12    ((1,[]), (1,[]))] }
13    val findParent = R.make { name = "findParent",
14    redex = makeBR redexParent,
15    react = reactParent,
16    inst = instParent,
17    info = info }
18    val BRsystemParent = makeBR currentSystem
19    val mtsd = M.matches { agent = BRsystemParent , rule =findParent }
20    val xNew = if ((lzLength mtsd) = "0") then ""
21               else
22                let
23                 val testalteragent' = M.unmk (LazyList.lzhd mtsd)
24                 val testalteragent'_par = #parameter(testalteragent')
25                 val x1 = Bdnf.unmkDR testalteragent'_par
26                 val x2 = #Ps(x1)
27                 val x3 = hd x2
28                 val x4 = (B.toString o B.simplify o Bdnf.unmk) x3
29                in
30                 x4
31                end
32   in
33    xNew
34   end
```

FIGURE 5-38: FUNCTION `findParent`.

To summarize, the function `findParent` uses a reaction rule that does not change the state of the WORLD layer to access information. This is achieved by having the same redex and

reactum for the reaction rule. This reaction rule is then passed on to the matching algorithm and it returns parameters from which the location can be extracted.

iv) `findChild`: This function finds one of the child locations of a location. We have made the simplifying assumption that because all locations in our model cache backup devices, we only need to find the first child rather than all children in order to find a supporting device for a specific service. Once again, we consider the state of the WORLD in Figure 5-34 which we reproduce in Figure 5-39. Suppose, we wish to find the child of location 1.

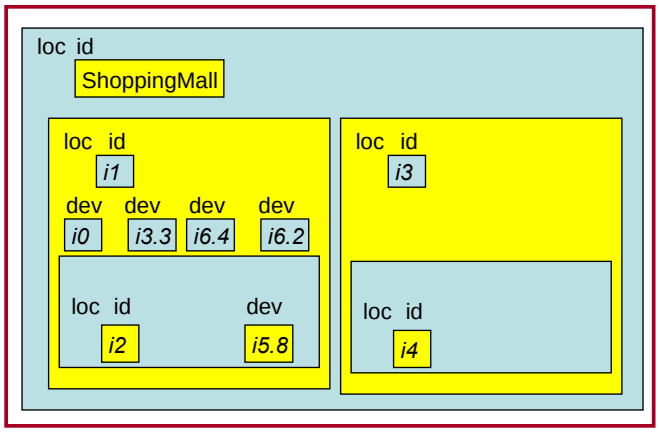

FIGURE 5-39: INITIAL STATE OF WORLD BIGRAPH.

We use the reaction rule shown in Figure 5-40. Notice from the figure that both the redex and reactum are the same Bigraphs. We pass the redex with `location = "1"` together with the WORLD layer Bigraph of Figure 5-39 to the matching algorithm.

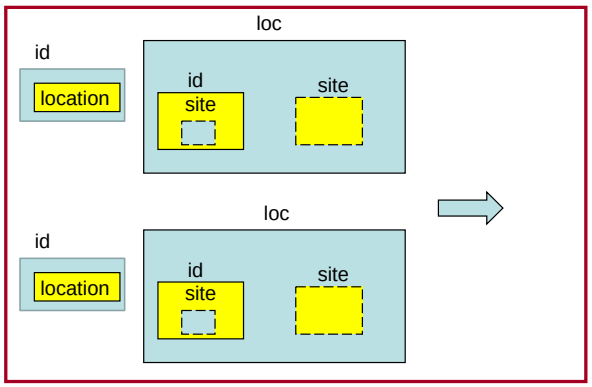

FIGURE 5-40: REACTION RULE WITH THE SAME REDEX AND REACTUM.

The context and the parameter that the matching algorithm returns are shown in Figures 5-41 and 5-42 respectively. Notice that the 'don't care' (site) of the reaction rule in Figure 5-40 is

'filled up' by the parameter in the Figure 5-42. Two Bigraphical parameters corresponding to the two sites in the redex of the reaction rule are returned. In Figure 5-42, the two Bigraphical parameters that have been returned are shown within dashed boxes. These dashed boxes represent the root outer interfaces of the Bigraphs as discussed in chapter 2. We can now extract out the first Bigraph from these two parameters.

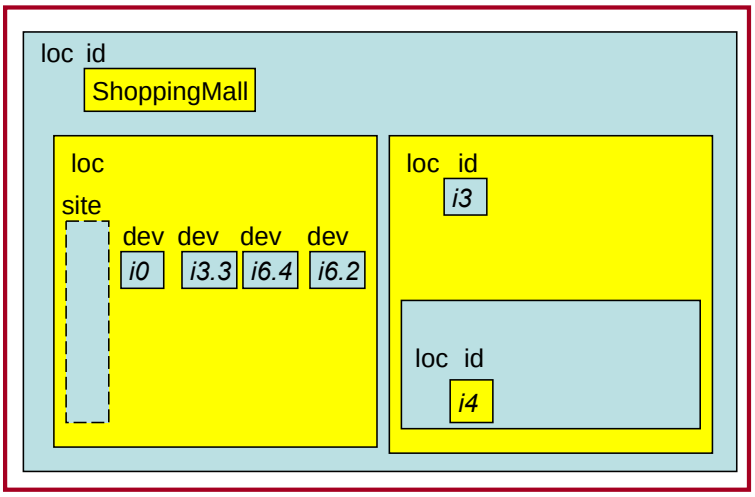

FIGURE 5-41: CONTEXT RETURNED BY THE MATCHING ALGORITHM.

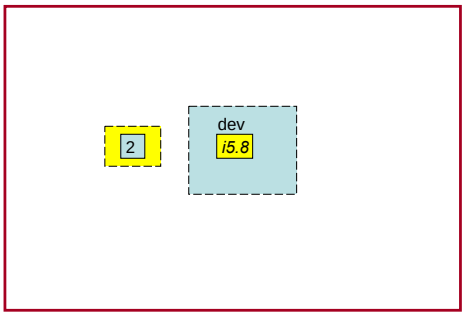

FIGURE 5-42: PARAMETER RETURNED BY MATCHING ALGORITHM.

We now discuss the code of this function shown in Figure 5-43.

The input parameters for the function are the id of the location whose child we wish to find (`location`) and the current state of the two layer model (`currentSystem`). The function returns the string representation of the id of the first of the child locations it finds.

We first declare the structure of the redex in line 3 and reactum in line 5. Next, in lines 7 and 8 respectively, we specify that the number of the sites in redex and reactum is two each (See Figure 5-40). We map the sites of the reactum to the sites of the redex in lines 9 to 11. We use the `make` function to construct the reaction rule in lines 12 to 16. Here, the structure R = BG.Rule , where 'Rule' is the abstract data type provided by BPL Tool for constructing rules. In line 17, we then convert the Bigraph that has been passed to `findChild` function from a BGVal to BDnf using Elsborg's code (Elsborg, 2009). Next, we compute a lazy list of the matches of redex with the Bigraph that needs to be rewritten using Elsborg's code (Elsborg, 2009) in line 18 to 19. Notice that we expect only one match because the parameter `location` passed to `findChild` is unique. In line 20, we calculate the length of the list containing the matches. Next, from lines 21 to 32, we use Elsborg's code (Elsborg, 2009) to define a curried function `locationIdentityStringList` that takes in a BDnf representation of a Bigraph and a lazy list of matches as input parameters and outputs the string representing the id of the child location. Within the function, in line 23 we extract the expected single match as the head of the lazy list. Next, in line 24, we extract only the parameters from the match (since a match contains both the context and the parameters). Then, in line 25, we define a function `peel` that uses BPL Tool's functions to convert a Bigraph that is in the BDnf form into a string. Next, in line 26, we deconstruct the parameter (which is of BDnf type) into its constituent link and place Bigraphs. Since we do not use links in our modeling, the constituent link returned will be empty and we will only have a list of place Bigraphs. In line 27, we extract the list of place graphs from the deconstructed parameter. Then, in line 28, we access the first Bigraph in the list as it will represent the child location's id. We use the `peel` function to convert this Bigraph into a string and then return this string in line 31. From lines 33 to 40, we use an if-else expression to construct the child location's identity string. In line 33, we write a guard condition and return an empty string if the length of the list containing the matches is zero. Line 34 starts the else branch of the expression. In line 36, we use the function that we defined namely `locationIdentityStringList` to return the string representing the child location's id to the variable `locIdStringList`. We then return this variable `locIdStringList` in line 42.

```
1  fun findChild(location,currentSystem) =
2   let
3    val redexLocateChild =
4    S.`|`(S.o(id,i(location)),S.o(loc,S.`|`(S.o(id,site),site)))
```

```
5    val reactLocateChild =
6    S.`|`(S.o(id,i(location)),S.o(loc,S.`|`(S.o(id,site),site)))
7    val redex_innerface_LocateChild = Iface.m 2
8    val react_innerface_LocateChild = Iface.m 2
9    val instLocateChild = Inst.make { I = redex_innerface_LocateChild,
10   J = react_innerface_LocateChild,maps = [((0,[]),(0,[])),((1,[]),
11   (1,[]))]}
12   val locateChild = R.make { name = "locateChild",
13   redex = makeBR redexLocateChild,
14   react = reactLocateChild,
15   inst = instLocateChild,
16   info = info }
17   val BRsystemLocateChild = makeBR currentSystem
18   val mtsd = M.matches { agent = BRsystemLocateChild , rule =
19   locateChild }
20   val len = lzLength mtsd
21   fun locationIdentityStringList agent m =
22    let
23     val agent' = M.unmk (LazyList.lzhd m)
24     val agent'_par = #parameter(agent')
25     fun peel x = (B.toString o B.simplify o Bdnf.unmk) x
26     val x1 = Bdnf.unmkDR agent'_par
27     val x2 = #Ps(x1)
28     val x3 = hd x2
29     val x4 = peel x3
30    in
31     x4
32    end
33    val locIdStringList = if (len="0")then ""
34                          else
35                           let
36                            val avariable = locationIdentityStringList
37                            BRsystemLocateChild mtsd
38                           in
39                            avariable
40                           end
41   in
42    locIdStringList
43   end
```

FIGURE 5-43: FUNCTION `findChild`.

In conclusion, the function `findChild` uses a reaction rule that does not change the state of the WORLD layer to access information. This is achieved by having the same redex and reactum

for the reaction rule. This reaction rule is then passed on to the matching algorithm and it returns parameters from which the location can be extracted.

v) `newFindParticipatingDevice`: We use this function to find the id of a device whose service is one of the services participating in the composition. This device will be stored in the Bigraphical array `compositionDevices`. We pass the `serviceId` of the service being offered by this device to the function `newFindParticipatingDevice`. The function returns the `deviceId` of the appropriate device. Consider the state of the Bigraphical array shown in the Figure 5-44. Suppose that we wish to find which device is offering service 2.

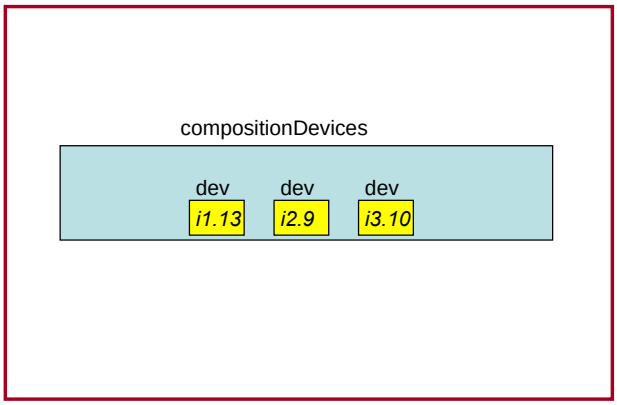

FIGURE 5-44: STATE OF THE BIGRAPHICAL ARRAY

We use the reaction rule shown in Figure 5-45. Notice from that figure that both the redex and reactum are the same Bigraphs. We pass the redex with `serviceId = 2` together with the Bigraph containing the Bigraphical array `compositionDevices` of Figure 5-44 to the matching algorithm. Notice that we do not specify the device Id of the device in the reaction rule because it is an unknown and we want the matching algorithm to return the parameter containing that information.

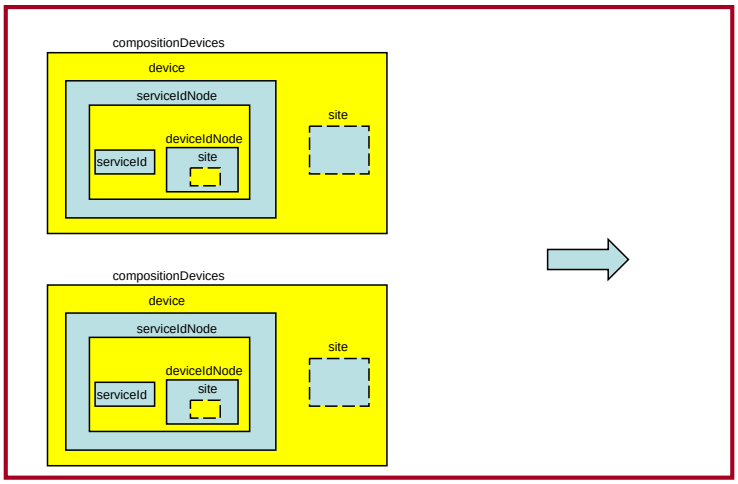

FIGURE 5-45: REACTION RULE WITH THE SAME REDEX AND REACTUM

FIGURE 5-46: CONTEXT RETURNED BY THE MATCHING ALGORITHM

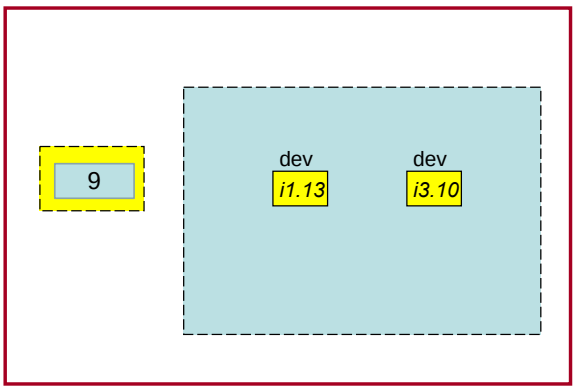

FIGURE 5-47: PARAMETERS RETURNED BY THE MATCHING ALGORITHM

The context returned by the matching algorithm is the site shown in Figure 5-46. The parameters returned by the matching algorithm are shown in Figure 5-47. There are two parameters in the Figure 5-47 corresponding to two sites in the reaction rule. The first parameter is an atomic node of zero arity encapsulating the ordinal number part of a device's id. The second parameter is a root node. Nested within this root node are the two other devices present in the `compositionDevices` Bigraphical array in the Figure 5-44. In the Figure 5-47, the two Bigraphical parameters that have been returned are shown within dashed boxes. These dashed boxes represent the root outer interfaces of the Bigraphs as discussed in chapter 2. We can now extract the atomic node from the parameters, and return it as a string.

We now discuss the code for function `newFindParticipatingDevice` shown in Figure 5-48. The input parameters for the function are the id of the service (`serviceId`) and the current state of the two layer model (`currentSystem`). The function returns the string representation of the id of the device.

Firstly, we declare the structure of the redex in line 3 to 5 and reactum in line 6 to 8. Notice that both of them represent the same Bigraph. Notice too that there is a site for the ordinal number inside the devIdNode. We want the matching algorithm to return the Bigraphical representation of the ordinal number as part of the parameter that it will return. Next, in lines 9 and 10 respectively, we specify that the number of the sites in redex and reactum is two each. We map the sites of the reactum to the sites of the redex in lines 11 to 14. We then use the `make` function to construct the reaction rule in lines 15 to 19. Here, structure R = BG.Rule , where 'Rule' is the abstract data type provided by BPL Tool for constructing rules. In line 20, we convert the Bigraph that has been passed to `newFindParticipatingDevice` function from a BGVal to BDnf using Elsborg's code (Elsborg, 2009). Next, we compute a lazy list of the matches of redex with the Bigraph that needs to be rewritten using Elsborg's code (Elsborg, 2009) in lines 21 and 22. Notice that we expect only one match because the parameter `serviceId` passed to `newFindParticipatingDevice` is unique. From lines 23 to 33 we define a function `getNodeLabel`. This function takes the type 'match' of the BPL Tool as its input, extracts out the Bigraphical parameter and returns it as string. In lines 25 to 29, we use BPL Tool's functions to extract out the Bigraph representing the parameter returned by the matching algorithm. In line 30, we convert the BGVal representation of a Bigraph to a string representation. Finally, we return this sting representation of the Bigraphical parameter in line 32.

On line 34, we pass the function `getNodeLabel` and the `mtsd` lazy list to the curried function `LazyList.lzmap` provided by the BPL Tool. The function `LazyList.lzmap` applies our function `getNodeLabel` to each element of the lazy list `mtsd` and returns a lazy list

consisting of the string representations of the BDnf representation of the ordinal numbers of the devices. Then in line 35, we convert the lazy list `aLazyList` into a list. On line 36, we define a curried function `prefixServiceId` which takes as inputs the string representation of the `serviceId` and the ordinal number of the device and constructs a string representing the `deviceId`. In line 37, we write a guard condition and return an empty string if the length of the list containing the matches is zero. Line 38 starts the else branch of the expression. In line 40, we extract the head of the string list since it contains the string representation of the ordinal number of the device. Then in line 42, we call our previously defined function `prefixServiceId` to construct the full device id and return it to a variable `testreturn`. Finally, in line 45, we return `testreturn`.

```
1  fun newFindParticipatingDevice(serviceId, currentSystem) =
2   let
3    val redexLocateDevice =
4    S.o(compositionDevices,S.`|`(S.o(device,S.o(serviceIdNode,S.`|`
5    (i(serviceId),S.o(devIdNode,site)))),site))
6    val reactLocateDevice =
7    S.o(compositionDevices,S.`|`(S.o(device,S.o(serviceIdNode,S.`|`
8    (i(serviceId),S.o(devIdNode,site)))),site))
9    val redex_innerface_LocateDevice = Iface.m 2
10   val react_innerface_LocateDevice = Iface.m 2
11   val instLocateDevice = Inst.make { I
12   =redex_innerface_LocateDevice,
13   J = react_innerface_LocateDevice, maps = [((0,[]), (0,[])),
14   ((1,[]), (1,[]))] }
15   val locateDeviceInArray = R.make { name = "locateDeviceInArray",
16   redex = makeBR redexLocateDevice,
17   react = reactLocateDevice,
18   inst = instLocateDevice,
19   info = info }
20   val BRsystemLocateDevice = makeBR currentSystem
21   val mtsd = M.matches { agent = BRsystemLocateDevice , rule =
22   locateDeviceInArray }
23   fun getNodeLabel(aNode) =
24    let
25     val testalteragent' = M.unmk (aNode)
26     val testalteragent'_par = #parameter(testalteragent')
27     val x1 = Bdnf.unmkDR testalteragent'_par
28     val x2 = #Ps(x1)
29     val x3 = hd x2
30     val x4 = (B.toString o B.simplify o Bdnf.unmk) x3
31    in
32     x4
```

```
33    end
34   val aLazyList = LazyList.lzmap getNodeLabel mtsd
35   val aList = LazyList.lztolist aLazyList
36   fun prefixServiceId aserviceId x = aserviceId ^ "." ^ x
37   val testreturn = if ((lzLength mtsd) = "0") then ""
38                    else
39                     let
40                      val locIdString = hd(aList)
41                     in
42                      prefixServiceId serviceId locIdString
43                     end
44  in
45   testreturn
46  end
```

FIGURE 5-48: FUNCTION `newFindParticipatingDevice`.

Summing up, in `newFindParticipatingDevice` function we define a reaction rule such that the BPL Tool's matching algorithm returns the ordinal number of the device that supports a specific service and is contained in the Bigraphical array `compositionDevices`. We extract that Bigraphical representation of the ordinal number, convert it into a string, construct the correct id of the device and return the resulting string.

vi) `constructSeviceTree`: In this function, we return a list of all the services participating in the composition. The BPL Tool's matching algorithm is used to traverse the tree representing the structure of the service composition at the SCA layer.

Consider the state of the SCA layer in Figure 5-1 which we reproduce in the Figure 5-49. We wish to find all the services participating in the composition. We use the reaction rule shown in Figure 5-50. Notice that as is the case with other functions of this section, both the redex and reactum are the same Bigraphs. Moreover, we do not specify the service id in the reaction rule. As a result, for the SCA layer shown in Figure 5-49, there will be four matches corresponding to four service ids. Each match consists of a context and a parameter. The context and parameter for one of the four matches is shown in Figures 5-51 and 5-52.

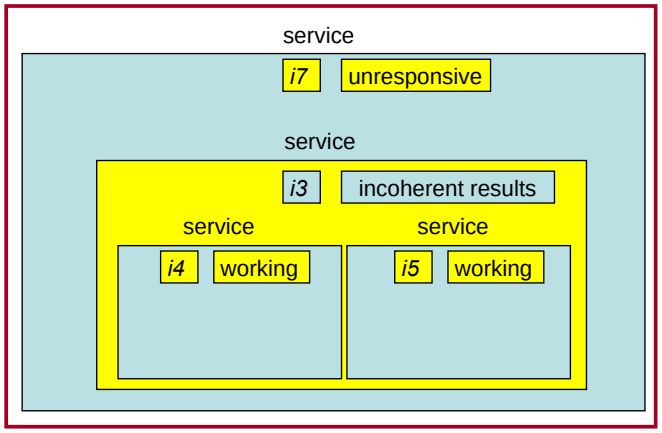

FIGURE 5-49: THE STATE OF THE SCA LAYER.

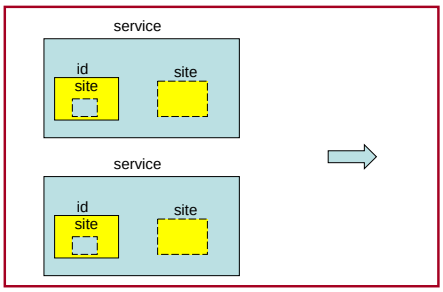

FIGURE 5-50: REACTION RULE WITH THE SAME REDEX AND REACTUM.

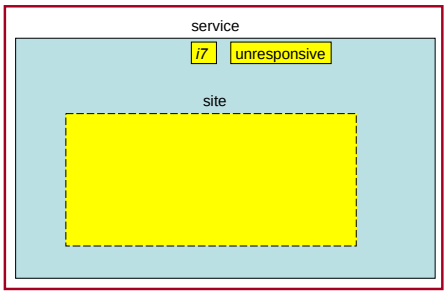

FIGURE 5-51: ONE OF THE FOUR CONTEXTS RETURNED BY THE MATCHING ALGORITHM.

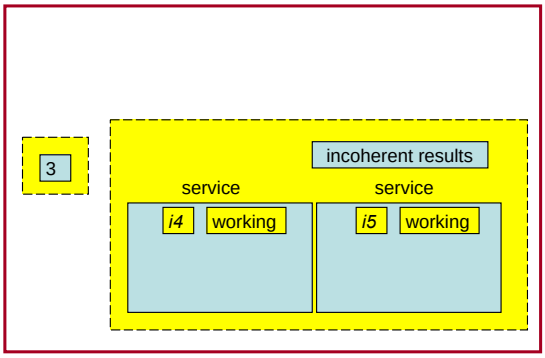

FIGURE 5-52: ONE SET OF TWO PARAMETERS RETURNED BY THE MATCHING ALGORITHM OUT OF FOUR SETS.

There are two parameters in the Figure 5-52 corresponding to two sites in the reaction rule. The first parameter is an atomic node of zero arity encapsulating a service's id. The second parameter is a root node. Nested within this root node are the two other services present within service 3. Also nested is a third Bigraph representing the state of service 3. In the Figure 5-52, the two Bigraphical parameters that have been returned are shown within dashed boxes. These dashed boxes represent the root outer interfaces of the Bigraphs as discussed in chapter 2. We can now extract the atomic node from the parameters, and return it as a string.

We now discuss the code of the function `constructSeviceTree` shown in Figure 5-53. Firstly, we declare the structure of the redex in line 3 and reactum in line 4. Notice that as has been the case in this section, both of them represent the same Bigraph. Next, in lines 5 and 6 respectively, we specify that the number of the sites in redex and reactum is two each. We map the site of the reactum to the site of the redex in lines 7 to 9. We then use the `make` function to construct the reaction rule in lines 10 to 14. Here, structure R = BG.Rule , where 'Rule' is the abstract data type provided by BPL Tool for constructing rules.

In line 15, we convert the Bigraph that has been passed to `constructServiceTree` function from a BGVal to BDnf using Elsborg's code (Elsborg, 2009). Next, we compute a lazy list of the matches of redex with the Bigraph that needs to be rewritten using Elsborg's code (Elsborg, 2009) in lines 16 and 17. Notice that we expect as many matches as the number of service ids. From lines 18 to 28 we define a function `getNodeLabel`. This function takes the type 'match' of the BPL Tool as its input, extracts out the Bigraphical parameter and returns it as string. In lines 21 to 24, we use BPL Tool's functions to extract out the Bigraph representing the parameter returned by the matching algorithm. In line 25, we convert the BGVal representation of a Bigraph to a string representation. Finally, we return this sting representation of the Bigraphical parameter in line 27. On line 29, we pass the function `getNodeLabel` and the `mtsd` lazy list

to the curried function `LazyList.lzmap` provided by the BPL Tool. The function `LazyList.lzmap` applies our function `getNodeLabel` to each element of the lazy list `mtsd` and returns a lazy list consisting of the string representations of the BDnf representation of the ordinal numbers of the devices. Then in line 30, we convert the lazy list `aLazyList` into a list. We return this list in line 32.

```
1  fun constructServiceTree(system) =
2   let
3    val redexTraverse = S.o (service,S.`|`(S.o(id,site),site))
4    val reactTraverse = S.o (service,S.`|`(S.o(id,site),site))
5    val redex_innerface_traverse = Iface.m 2
6    val react_innerface_traverse = Iface.m 2
7    val instTraverse = Inst.make { I = redex_innerface_traverse,
8    J = react_innerface_traverse,maps = [((0,[]), (0,[])),((1,[]),
9    (1,[]))] }
10   val serviceTreeTraversal = R.make { name = "serviceTreeTraversal",
11   redex = makeBR redexTraverse,
12   react = reactTraverse,
13   inst = instTraverse,
14   info = info }
15   val BRserviceTree = makeBR system
16   val mtsd = M.matches { agent = BRserviceTree , rule =
17   serviceTreeTraversal }
18   fun getNodeLabel(aNode) =
19    let
20     val testalteragent' = M.unmk (aNode)
21     val testalteragent'_par = #parameter(testalteragent')
22     val x1 = Bdnf.unmkDR testalteragent'_par
23     val x2 = #Ps(x1)
24     val x3 = hd x2
25     val x4 = (B.toString o B.simplify o Bdnf.unmk) x3
26    in
27     x4
28    end
29   val aLazyList = LazyList.lzmap getNodeLabel mtsd
30   val aList = LazyList.lztolist aLazyList
31  in
32   aList
33  end
```

FIGURE 5-53: FUNCTION `constructServiceTree`.

Summing up, the function `constructServiceTree` traverses through the structure of the service composition and returns a string list of all the participating services.

Thus, we have seen that one of the ways that we have explored to write functions that access information from the PGM-like model, is to design reaction rules with the same redex and reactum to be able to utilize the matching algorithm.

5.3.3 SECTION SUMMARY

In this section, we have discussed our implementation of two classes of functions that have used the matching algorithm- functions that modify the model and functions that access information from the model. We have shown a way to have the matching algorithm return a single match by utilizing concepts of abstraction by parameterization to write our functions that encapsulate the reaction rules. This has also given us the capability to intensionally generate infinitely many reaction rules.

Thus, our implementation of runtime model can deal with ill-structured run-time phenomenon with no pre-determined order of runtime events. Moreover, we can express infinitely many reconfigurations at runtime using our reaction rules since there are an infinite number of the values for the parameters of the functions that encapsulate the reaction rules.

5.4 FUNCTIONS TO ENCAPSULATE ADAPTATION LOGIC AND SIMULATE TEST RUNS

Our two-layered model captures those effects of volatility that result in the generation of service faults identified by Chan (Chan et al., 2007a). As discussed in Chapter 4, we assume that a Service Component Architecture (SCA) (Marino and Rowley, 2010) like description of all the services and the service composition is being maintained by a system outside our system boundary.

As discussed in Chapter 2, according to Waddington and Lardieri (Waddington and Lardieri, 2006), *"Models should abstract selected elements of the implemented complex system"* rather than *"replicate abstractions that programming languages provide"*.

In our implementation of model at runtime, 'the implemented complex system' is the SCA specification of a service composition running on a mobile device and the 'selected elements' that our model abstracts are the services and their fault states that an SCA specification of a service participating in the composition can suffer from. Notice as discussed in Chapter 4, the implemented system and our model are *different* entities.

As discussed in Chapter 4, the SCA layer is one of two layers in our model. The other layer is the WORLD layer. The SCA layer models the SCA specification of a service composition

running on a mobile device which is the volatile system. Thus, the SCA layer represents the internal view of the system. On the other hand, WORLD layer models the cached location and identity of devices offering service and also the location of the user's device. Thus, the WORLD layer represents the external view, that is, the environment in which the system is located.

At the SCA layer, each 'observed effect' of a fault in the service participating in the composition is modelled as a reaction rule. At the WORLD layer, there is a reaction rule for caching and another one for un-caching of the identity of a device offering a service. A third reaction rule at the WORLD layer models any device (we will use this reaction rule for the user's device `dev0`) moving from one ambient to another.

Our two-layered model captures volatility inherent in our scenario by modelling both the system and the environment view of the effects of volatility. We discuss the implementation of our functions in sub-section 5.4.1.

5.4.1 IMPLEMENTATION OF THE FUNCTIONS

Each of the reaction rules at the SCA layer can be *independent* of the reaction rule at the WORLD layer that models the user's device moving from one ambient to another- the reaction rules at the SCA layer might or might not be triggered simultaneously with the latter. For example, a service forming part of the service composition might develop a fault corresponding to a reaction rule of the SCA layer without the user's device having moved anywhere. On the other hand, a service forming part of the service composition might develop a fault corresponding to a reaction rule of the SCA layer *because* of the user's device having moved to another ambient. Another possibility is where a user's device could have moved to another ambient without triggering any fault in the service forming part of the service composition corresponding to a reaction rule of the SCA layer.

We have, therefore, written two separate functions that encapsulate adaptation logic and simulate our test runs: `SCAFaultScript` which models the case where a reaction rule at the SCA layer (System's view of effects of volatility) has been triggered and another function `changeAmbientScript` which models the case where the reaction rule for the user's device having changed its ambient has been triggered (Environment's view of effects of volatility). To model the case where service develops a fault *and* user moves from one ambient to another, we call these two functions one after another in our test runs. We now discuss each of these functions and their supporting functions.

5.4.1.1 FUNCTION DEALING WITH SYSTEM'S VIEW OF EFFECTS OF VOLATILTY

The function `SCAFaultScript` encapsulates adaptation logic and simulates test runs for the case where an 'observed effect' of Table 4-2, column 2, has occurred, triggering a corresponding reaction rule at the SCA layer. The function uses information stored in the PGM-like model to implement the adaptation logic by sending out appropriate adaptation commands. It, then, updates the information stored in the model.

The adaptation logic as shown in Table 5-1 is as follows: We unbind the faulty service from the service composition. Internally, we also un-cache the service's device in the WORLD layer and change the state of the service as appropriate to one of the five fault states in the SCA layer. Next, we send out a command to bind a new service to the composition. Internally, this service's device and its location have already been pre-fetched in the cache. Also, we change the state of service back to 'working' (See Chapter 4). Finally, we send out a command to pre-fetch the identity of a device that offers an alternative backup service in the new ambient. Note that we have made the simplifying assumption that there are backup devices available in every ambient and thus we can always find a suitable service to replace a malfunctioning one.

The supporting functions for this function that we discuss are `SCANewState`, `repairCompositionPolicy`, and `getDeviceList`.The input parameters for `SCAFaultScript` function are: `newDeviceId` (the Id of an alternative device that is offering a service similar to the one that has suffered a fault), `newDeviceAmbient` (either the current location or the parent or child location where the alternative device has been found) `serviceId` (the Id of the faulty service), `initialState` (the initial state of the service), `finalState` (the final faulty state of the service), `system` (the BGVal representation of our two-layered model), and `stringPtr2` which is an SML reference type that will point to the following string value once the call to `SCAFaultScript` returns: `currentAmbient` (this represents the current ambient where the user's device and hence the user is located), `serviceId` (this represents the id of the service whose device developed the defect), `newDeviceId` (this is the new device's id), `deviceId` (this is the defective device's id), and `defectiveDeviceLocation` (this is the defective device's location). These string values will form the command that will be sent by our Bigraphical model at runtime to an external Android device that generated this fault (See next chapter for more details). Notice that passing the parameters `newDeviceId` and `newDeviceAmbient` to the function `SCAFaultScript` simulates the fact that there is an external system that sends us this information, which we then use to exercise our adaptation logic.

We now discuss the function's code (Figure 5-54) in detail. In line 6, we change the state of service from 'working' to a state that represents one of the 'observed effects' of Table 4-2. We

assume that the event that generates this function includes the information about the new state of the service. Next, in lines 7 to 8, we call the function `repairCompositionPolicy` (described later). This function implements the policy on how to respond when a reaction rule modelling one of the 'observed effects' of Table 4-2 is triggered (See a detailed discussion of this function below). Notice that we pass the reference `stringPtr2` to `repairCompositionPolicy` which changes the string that is being referenced by stringPtr2 to: "`newDeviceId, deviceId, defectiveDevicelocation`". In line 9, we call `locateDevice` function on our PGM-like model to retrieve where the device with the device Id `i0` (user's device running the service composition) is located. Next, in line 10, we construct the command to pre-fetch another device's Id that is offering an equivalent service in a 'nearby' ambient. We define 'nearby' to mean any device in either the current or parent or child ambient. For reducing complexity in our implementation, we have defined 'nearby' in such a fashion. In line 13, we point the reference `stringPtr2` to the command string that we constructed on line 10. Finally, in line 14, we simulate the caching of a new device. This new device's Id and its location are generated by the event of a system (An Android device, see next chapter for details) outside our system boundary sending us the new device Id and its location. We assume that a system outside our system boundary searches for an 'equivalent' service, finds a device offering it in a 'nearby' ambient (either current or the parent or child ambient) and sends our system the device's Id and location. Because, this search is a pre-fetch, our system does not have to wait for a replacement of a faulty service since a faulty service is always replaced by an equivalent service that has already been cached in our model.

```
1  fun
2  SCAFaultScript(newDeviceId,newDeviceAmbient,serviceId,initialState,
3  finalState,system ,stringPtr2) =
4  let
5  val system1 =
6  SCANewState(serviceId,initialState,finalState,system)
7  val system2 =
8  repairCompositionPolicy(serviceId,finalState,system1,stringPtr2)
9  val currentAmbient = locateDevice("0",system2)
10 val testString1 = currentAmbient^ ","^ serviceId ^","^
11 (!stringPtr2)
12 in
13 stringPtr2 := testString1;
14 deviceAppears(newDeviceId,newDeviceAmbient,system2)
15 end
```

FIGURE 5-54: FUNCTION SCAFaultScript.

Summing up, the function `SCAFaultScript` is used by us to respond to the event where one of the five 'observed effects' of Chan et al. (Chan et al., 2007a) has occurred. It also includes adaptation logic to decide which commands to send out and what modifications to make in our PGM-like model.

We wish to point out some limitations of the adaptation logic that we have just discussed. We assume that a system outside our system boundary can always find a suitable device, that there is no delay in the system outside our system boundary returning this information to us, and that the device that is returned to us is always a device unknown to us. Notwithstanding these limitations, we would want to test in the next chapter if the Bigraphical model at runtime can still be in-sync with the running system and the world.

Two supporting functions appear in `SCAFaultScript` function: `SCANewState` and `repairCompositionPolicy`. We now describe implementation of each of these functions.

`SCANewState`: This function changes the state of a service. The input parameters for this function are: `serviceId` (the Id of the service), `initialState` (the initial state of the service), `finalState` (the final state of the service), `and system` (the BGVal representation of our two-layered model). We discuss the function's code in detail (See Figure 5-55). In lines 3 to 4, we construct the reaction rule to change the state of a service with the appropriate Id of the service, its initial state and its final state. Then in line 6 we apply the reaction rule to the system passed as an input parameter to the function.

```
1  fun SCANewState(serviceId,initialState,finalState,system) =
2   let
3    val changedState =
4    constructStateChange(serviceId,initialState,finalState)
5   in
6    changeSystem(system,changedState)
7   end
```

FIGURE 5-55: FUNCTION SCANewState.

In conclusion, the function `SCANewState` is used by us to change the state of a service.

`repairCompositionPolicy`: This function implements the policy on how to respond when a reaction rule modelling one of the 'observed effects' of Table 4-2 is triggered. Essentially, this policy is to replace the faulty service by an equivalent service being offered by a device in a 'nearby' ambient. As discussed earlier, we define 'nearby' to mean any device in the current or parent or child ambient. Notice that, the alternative device's Id has already been pre-fetched and

cached in our PGM-like model. Because of this caching, we do not need to search for such a device at the precise time that we discover that we need to use it.

The input parameters for this function are: `serviceId` (the Id of the faulty service), `serviceState` (the faulty state of the service), and `system` (the BGVal representation of our two-layered model) and `stringPtr2` which is an SML reference type that will point to the following string value once the call to `repairCompositionPolicy` returns: `newDeviceId` (this is the new device's id), `deviceId` (this is the defective device's id), `defectiveDeviceLocation` (this is the defective device's location). These string values will form the command that will be sent by our Bigraphical model at runtime to an external Android device that generated this fault (See next chapter for more details).

The function `repairCompositionPolicy` returns the modified system.

Now, we discuss the implementation of the function in detail (Figure 5-56). As discussed earlier, we have used a Bigraphical array `compositionDevices` to store the device id of those devices whose services are participating in the composition. In line 4, we use the function `newFindParticipatingDevice` function to find the id of the device whose service has developed a defect. Next, in line 5, we find the location of the defective device. In line 6, we start the construction of a string that we will send as a command to a system outside our system boundary (An Android device-see next chapter). This string on line 6 at this stage includes the defective device's id and its location. Later more information will be prefixed to this string. In lines 7 and 8, we un-cache the defective device. Then, in line 9, we save the device Id of the defective device in a new variable `oldDeviceId`. In line 10, we 'interrogate' our model to find the current location of the device running the service composition. As discussed earlier, the id of this device is '0'. In line 11, we use the function `getDeviceList` written by us to add to the composition an equivalent service being offered by another device. If no such device is found in the current ambient, the function tries the parent ambient and then the child ambient. We assume that we have already pre-fetched and cached those device Ids and their locations that offer equivalent service. The function `getDeviceList` returns a list containing devices offering an equivalent service. In lines12 and 13, we extract the first device that is in the list. In lines 14 to 16, we update the Bigraphical array binding the service to the new device, and construct `testString2` which adds the string representation of `newDeviceId` to `testString1`. This new variable `testString2` will form part of the command sent to a system outside our system boundary. In line 18, we point the reference `stringPtr2` to the command string that we constructed on line 16. Finally, in line 19, we trigger an SCA rule to change the state of the defective service back to 'working' and return the resulting system.

```
1  fun repairCompositionPolicy(serviceId,serviceState,system,stringPtr2)
2   =
3   let
4    val deviceId = newFindParticipatingDevice(serviceId,system)
5    val defectiveDeviceLocation = locateDevice(deviceId,system)
6    val testString1 = deviceId ^ "," ^ defectiveDeviceLocation
7    val newSystem =
8    deviceDisappears(deviceId,defectiveDeviceLocation,system)
9    val oldDeviceId = deviceId
10   val currentAmbient = locateDevice("0",newSystem)
11   val deviceList = getDeviceList(serviceId,currentAmbient,newSystem)
12   val newDeviceId = if(not(deviceList = nil)) then hd(deviceList)
13                     else ""
14   val newSystem3 = deviceLeavesComposition(oldDeviceId,newSystem )
15   val newSystem2 = deviceJoinsComposition(newDeviceId,newSystem3)
16   val testString2 = newDeviceId ^ "," ^ testString1
17  in
18   stringPtr2 := testString2;
19   SCANewState(serviceId,serviceState,"working",newSystem2)
20  end
```

FIGURE 5-56: FUNCTION `repairCompositionPolicy`.

Summing up, the function `repairCompositionPolicy` replaces a faulty service with a service being offered by an alternative device in either the current or parent or the child ambient.

We now discuss the implementation of the function `getDeviceList` appearing above in Figure 5-56 in line 11.

`getDeviceList`: This function is used to add to the composition an equivalent service being offered by another device. If no such device is found in the current ambient, the function tries to find the device in the parent ambient and then the child ambient. We assume that we have already pre-fetched and cached those device Ids and their locations that offer equivalent service. The function returns a list containing devices offering an equivalent service.

Consider Figure 5-57. In lines 3 and 4, we use the identifier `deviceList0` to represent the list of all devices in current ambient offering the service with Id `serviceId` returned by the function `enumerateDevicesInShoppingMall`. Next, in line 5, we find the parent and in line 6, we find the child of the current ambient. In line 8, if the list `deviceList0` is not empty, we return it. Thereafter, in lines 11 and 12, since the list `deviceList0` is empty, we test if parent of the current ambient exists. If it does exist, we again call the function `enumerateDevicesInShoppingMall` passing to it the location Id of the parent ambient. The

resulting list is represented by the identifier `deviceList1`. If we find that there is no parent ambient, the list `deviceList` is `nil` in line 14. Then, in line 16, if the list `deviceList1` is not nil, we return it. Next, in line 18 and 19, since the `deviceList1` is empty, we test if a child ambient of the current ambient exists. If it does exist, we again call the function `enumerateDevicesInShoppingMall` passing to it the location Id of the child ambient. The resulting list is returned directly. Finally, in line 21, if there is no child ambient, we return an empty list.

```
1  fun getDeviceList(serviceId,currentAmbient,newSystem)=
2  let
3    val deviceList0 =
4    enumerateDevicesInShoppingMall(serviceId,currentAmbient,newSystem)
5    val parentAmbient = findParent(currentAmbient,system0)
6    val childAmbient = findChild(currentAmbient,system0)
7   in
8    if(not(deviceList0=nil)) then deviceList0
9    else
10    let
11    val deviceList1 = if(not(parentAmbient="")) then
12    enumerateDevicesInShoppingMall(serviceId,parentAmbient,newSystem)
13    else
14     nil
15    in
16     if(not(deviceList1=nil)) then deviceList1
17     else
18      if(not(childAmbient="")) then
19      enumerateDevicesInShoppingMall(serviceId,childAmbient,newSystem)
20      else
21       nil
22    end
23  end
```

FIGURE 5-57: FUNCTION `getDeviceList`.

To summarize, the function `getDeviceList` finds devices offering equivalent service in current ambient, failing which, it tries first the parent and then the child ambient.

Thus, we see that the function `SCAFaultScript` encapsulates our scenario where a service participating in the service composition running on a mobile device develops a fault.

5.4.1.2 FUNCTION DEALING WITH ENVIRONMENT'S VIEW OF EFFECTS OF VOLATILTY

The function `newChangeAmbientScript` encapsulates adaptation logic and simulates test runs for the case where the reaction rule for the user's device having changed its ambient has been triggered. The adaptation logic is as follows: It simulates an output command by sending out to a device outside our system boundary (An Android device – see next chapter) the following information in a string: a) The identity of the new ambient, b) A list of those services whose devices have not been cached in the WORLD layer of model in the new ambient c) Either the identity of the parent ambient or the child ambient of the new ambient where those services' devices have not been cached. It also updates the location of user's device in the model to the new ambient. This follows the logic discussed in Table 5-1.

We also discuss the following supporting functions for this function :

`newChangeAmbientOutputCommand`,

`newCreateServiceList`,

`filter`,

`testIfServiceNotSupported`,

and `preFetch`.

The input parameters for `newChangeAmbientScript` function are: `initialLocation` (the initial location of the user's device), `finalLocation` (the final location of the user's device), and `system` (the BGVal representation of our two-layered model) and `stringPtr` which is an SML reference type that will point to the following concatenated string values once the call to `newChangeAmbientScript` returns: the final location of the user's device, a list of services, and either the identity of the parent or the child ambient. These string values will form the command that will be sent by our Bigraphical model at runtime to an external Android device that generated this fault (See next chapter for more details).

The function returns the modified model. We now discuss the implementation of the function (see Figure 5-58). In line 6, the function `newChangeAmbientOutputCommand` (described later) points the SML reference to a string that will be sent out as command the details of which have already been discussed above. Finally, in line 9, the function returns the model with updated location of the user's device.

```
1  fun
2  newChangeAmbientScript(initialLocation,finalLocation,system,
3  stringPtr)=
4   let
5    val _ =
6    newChangeAmbientOutputCommand(initialLocation,
7    finalLocation,system, stringPtr)
8    in
9     changeAmbient("0",initialLocation,finalLocation,system)
10  end
```

FIGURE 5-58: FUNCTION changeAmbientScript.

To conclude, the function newChangeAmbientScript is used by us to simulate the event where user's device has moved from one ambient to another. It also includes adaptation logic to decide which commands to send out and what modifications to make in our PGM-like model.

newChangeAmbientOutputCommand function: We now describe the implementation of the function newChangeAmbientOutputCommand which appears in line 5, in Figure 5-58. This function sends out the list of those services in a 'nearby' ambient whose devices have not been cached in the model to a system outside our system boundary. It also sends the identity of the new ambient. As discussed earlier, we define 'nearby' to mean only device in the current or parent or child ambient (location) to simplify our implementation. Notice that if the user's device has moved down a node in the location tree, we need to list services from the new location's child ambient only. This is because the new location's parent ambient was our previous current ambient and so we already have a list of devices from that ambient. Similarly, if the user's device has moved up a node in the location tree, we need to list services from the new location's parent ambient only. This is because the new location's child ambient was our previous current ambient and so we already have a list of devices from that ambient in our model.

The input parameters for changeAmbientOutputCommand function are: initialLocation (the initial location of the user's device), finalLocation (the final location of the user's device), and system (the BGVal representation of our two-layered model) and stringPtr which is an SML reference type that will point to the following string value once the call to newChangeAmbientScript returns: the final location of the user's device, a list of services, and either the identity of the parent or the child ambient. These string values will form the command that will be sent by our Bigraphical model at runtime to an external Android device that generated this fault (See next chapter for more details).

We now discuss `newChangeAmbientOutputCommand` function in detail (See Figure 5-59). In lines 6 and 7, we define the function `concatString` that given a list of strings concatenates them together into one string with commas separating each individual string of the original list. In lines 8 and 9, we find the parent and one of the child locations of the final location. The identifier system0 is our original model- this works just as well here because locations remain immutable in our model. In lines 11 and 12, we check if we have moved down the location tree and whether a child location of the final location exists. In line 14, we use the function `newCreateServiceList` (explained below) to return a list of services whose devices have not been cached in this child location. In lines 15 and 16, we construct the string command that will be sent out to a system outside our system boundary. In line 18, we point the reference `stringPtr` to the string command that we have constructed. Then on line 19,we also printout the command to pre-fetch a new device's Id and cache its location in the current ambient. In the else branch from the lines 21 to 27, we construct an empty string because there is no child ambient.

In lines 28 to 30, we have not moved down, so we check if we have moved up the location tree and whether a parent location of the final location exists. In line 32, we use the function `newCreateServiceList` (explained below) to return a list of services whose devices have not been cached in this parent location. In lines 33 to 35, we construct the string command that will be sent out to a system outside our system boundary. In line 37, we point the reference `stringPtr` to the string command that we have constructed. Then on line 38, we also printout the command to pre-fetch a new device's Id and cache its location in the current ambient. In the else branch from the lines 40 to 45, we construct an empty string because there is no parent ambient.

From lines 46 to 52, we place a guard else statement that sends an empty string if our initial location is neither a parent nor a child.

```
1  fun
2  newChangeAmbientOutputCommand(initialLocation,finalLocation,
3  system, stringPtr)
4  =
5   let
6    fun concatString(nil) = ""
7    |concatString(x::xs) = x ^ ","^(concatString(xs))
8    val parentAmbient = findParent(finalLocation,system0)
9    val childAmbient = findChild(finalLocation,system0)
10  in
11   if(parentAmbient = initialLocation) then
12    if(not(childAmbient = "")) then
```

```
13     let
14      val listOfServices = newCreateServiceList(childAmbient, system)
15      val testString1 = concatString(listOfServices)
16      val testString2 = childAmbient^","^finalLocation^","^testString1
17     in
18      stringPtr := testString2;
19      preFetch(childAmbient)
20     end
21    else
22     let
23      val testString2 = ""
24     in
25      stringPtr := testString2;
26      print("\n")
27     end
28  else
29    if(childAmbient = initialLocation) then
30     if(not(parentAmbient = "")) then
31      let
32       val listOfServices = newCreateServiceList(parentAmbient,system)
33       val testString1 = concatString(listOfServices)
34       val testString2 = parentAmbient^","^finalLocation^",
35       "^testString1
36      in
37       stringPtr := testString2;
38       preFetch(parentAmbient)
39      end
40     else
41      let
42       val testString2 = ""
43      in
44       stringPtr := testString2;
45      end
46    else
47     let
48      val testString2 = ""
49     in
50      stringPtr := testString2;
51      print("\n")
52     end
53 end
```

FIGURE 5-59: FUNCTION changeAmbientOutputCommand.

To sum up, the function newChangeAmbientOutputCommand sends out a list of services whose devices have not been cached in any of the current, parent or child ambient.

We now discuss newCreateServiceList function appearing in lines 14 and 32 in the Figure 5-59.

newCreateServiceList : This function returns a list of services whose devices have not been cached in a given location in the model. The input parameters for this function are: ambient (the given location), and system (the BGVal representation of our two-layered model). We now discuss the function in detail (see Figure 5-60). In line 3, we use the function constructServiceTree to get a list of service ids as strings. Then, in line 5, we use the curried function filter and pass to it another curried function called testIfServiceNotSupported, and the parameters ambient, system and the list of services serviceList. From this list of services, we filter out a list of services that are not supported in the ambient and return the filtered list.

```
1  fun newCreateServiceList(ambient, system) =
2    let
3     val serviceList = constructServiceTree(system)
4    in
5     filter((testIfServiceNotSupported (ambient,system)),(serviceList))
6    end
```

FIGURE 5-60: FUNCTION newCreateServiceList.

To sum up, the function newCreateServiceList returns a list of services whose devices have not been cached in the given ambient.

We now discuss the function filter, which appears, on line 5 in the Figure 5-60. This has been taken from Ullman's textbook (Ullman, 1998). It takes as input a Boolean function and a list. It returns only those elements in the list that satisfy the Boolean function.

We now discuss the code of filter function shown in Figure 5-61 in detail. This function is recursive and line 1 defines the base case returning a nil if the pattern matches the Boolean function and nil. Lines 2 and 3 define the recursive case where the Boolean function is applied to the first element of the list and a recursive call to filter is applied to the rest of the list.

```
1  fun filter(P,nil) = nil
2  | filter(P,x::xs) =
3   if P(x) then x::filter(P,xs)
4   else filter(P,xs)
```

FIGURE 5-61: FUNCTION `filter`.

The function `testIfServiceNotSupported` that appears on line 5 in Figure 5-60 is discussed next. This is a curried predicate (Boolean function) to test if an ambient has no supporting device for a specific service. Consider Figure 5-62: In line 3, we use the function `enumerateDevicesInShoppingMall` to find if the specified service is being offered by any device in the `ambient` variable that has been passed into `testIfServiceNotSupported`. In line 6, we return true if the list returned in line 3 is nil. Other wise, in line 7, we return false.

```
1  fun testIfServiceNotSupported (ambient, system) serviceId =
2   let
3    val deviceList =
4    enumerateDevicesInShoppingMall(serviceId,ambient,system)
5   in
6    if (deviceList = nil) then true
7    else false
8   end
```

FIGURE 5-62: FUNCTION `testIfServiceNotSupported`.

Finally we discuss the function `preFetch` that appeared in lines 19 and 38 in Figure 5-59. This function prints out the ambient for which devices need to be pre-cached.

```
1  fun preFetch(currentAmbient) =
2   print("Pre-fetch a new device's Id TO SUPPORT EACH OF THE ABOVE
3   SERVICES and cache EACH DEVICE'S location in the ambient: " ^
4   currentAmbient ^ "\n"^"\n")
```

FIGURE 5-63: FUNCTION `preFetch`.

Summarizing, we see that the function `newChangeAmbientScript` encapsulates our scenario where the user moves from one ambient to another in a Shopping Mall.

5.4.2 SECTION SUMMARY

In this section, we have discussed implementation of functions that encapsulate adaptation logic and simulate test runs. We have written these functions to deal with two points of view-from an internal systems point of view and an external environmental point of view of effects of volatility. These functions have been used by us to simulate our test runs.

5.5 CONCLUSIONS

In this chapter, we pointed out the boundaries of our system implementation, the unused features of Bigraphs owing to the limitation of the BPL Tool, and the way we organized the input events and output commands of our system. We then showed how we have used abstraction by parameterization concepts to implement functions that modify the model or access information from them and to generate infinitely many rules intensionally. Finally, we showed how our reaction rules capture some of the effects of volatility on a service composition running on a mobile device and discussed our implementation of functions that encapsulate adaptation logic and simulate test runs.

In summary, our implementation serves as a proof of concept that Bigraphs can be used to express appropriate abstractions for a two-layered model at runtime for managing ubiquitous computing volatility. We have been able to combine two views- environment and the system into one model. Moreover, from a programming perspective, we have shown a way to implement PGM-like models as data structures by storing information in them and SML functions that access or modify the information that is stored in PGM-like models as algorithms. We have used well-established software engineering principles of abstraction and modularity to implement our system. Above all, we have discussed our code in enough detail in this chapter such that it could be independently re-implemented if the reader so wished. We also acknowledge that we un-cache devices only when there is an observed failure. Thus if a system outside our boundary fails to report a failure, our model will become out of sync with the running system. In the next chapter, we test if even with these restrictions, the system remains in sync with a running system and the movement of the user in a shopping mall.

6 A QUALITATIVE AND QUANTITATIVE EVALUATION OF THE BIGRAPHICAL MODEL AT RUNTIME

6.1 INTRODUCTION

So far in this thesis, we have analyzed the state of the art relevant to us in Chapter 2. Next in Chapter 3, we discussed the research question and its design implications. Then, we showed in Chapter 4 a way to tackle the volatility problem of ubiquitous computing systems using Bigraphs and models at runtime. Finally, in Chapter 5, we showed a way to use the BPL Tool to implement a two-layered model at runtime.

Now, in this chapter, we analyze our Bigraphical model both qualitatively and quantitatively. We show that because of the inefficiency of the matching algorithm of the BPL Tool, it is not practical to use a Bigraphical model at runtime that is built on top of the BPL Tool for realistic scenarios. This chapter thus presents an evaluation of the system that we built to answer the research question posed in Chapter 3.

We have organized this chapter as follows: In section 6.2, we qualitatively analyze our system by placing it in the context of the modeling dimensions of self-adaptive software systems. In section 6.3, we conduct a quantitative performance evaluation of the response times of our Bigraphical model at runtime. Finally, in section 6.4 we provide a summary of the chapter.

6.2 A QUALITATIVE DISCUSSION: PLACING OUR IMPLEMENTATION IN CONTEXT

Andersson et al. (Andersson et al., 2009) have presented a taxonomy of modeling dimensions of self adaptive software systems to *" provide engineers with a common set of vocabulary for specifying the self-adaptive properties under consideration and select suitable solutions"*. We qualitatively analyze our implementation of a model at runtime expressed with Bigraphs in the context of these modelling dimensions (shown in Table 6-1). Our goal is to identify any shortcomings in our design and to place it in the context of the work being done by the software engineering community.

In the taxonomy, each aspect of a system that is relevant for self-adaptation is described by a dimension (Cheng et al., 2009) . These dimensions are organized into four categories: Goals, Changes, Mechanisms and Effects. Our Bigraphical model at runtime and the service composition to which it is causally connected are our system for the purposes of the discussion in this section. We now describe our system's dimensions in each of these categories:

1) Goals: Goals are objectives the system under consideration should achieve (Andersson et al., 2009). Our system's goal is that the service composition should be running at all times despite volatility. We define each dimension of this category and the degree of each dimension that our system exhibits:

a) Evolution: This dimension captures whether goals can change within the lifetime of a system (Andersson et al., 2009). The goal of our system is static since it does not change within the lifetime of the service composition.
b) Flexibility: This dimension identifies if the goals are flexible in the way they are expressed (Andersson et al., 2009). Our system's goal is rigid- the service composition *must* continue working.
c) Duration: This dimension refers to the validity of a goal through the system's lifetime (Andersson et al., 2009) . The goal of our system is persistent since we want to keep the service composition running throughout its lifetime.
d) Multiplicity: This dimension is concerned with the number of goals a system may have (Andersson et al., 2009). Our system has a single goal namely to keep the service composition running in the face of a high rate of malfunctioning of services due to volatility.
e) Dependency: This dimension captures how goals are related to each other if a system has more than one goal (Andersson et al., 2009). Since our system has only a single goal, this dimension is not relevant to us.

2) Change: Change is the cause of adaptation (Andersson et al., 2009). The reason for change in our system is volatility which is an environment dependent variation. The definition of each dimension in this category and the degree of each of those dimensions that our system exhibits are:

a) Source: This dimension describes the source of change (Andersson et al., 2009). In our system, the source of change is volatility caused by the external environment.
b) Type: This dimension refers to the nature of change (Andersson et al., 2009).The type of change is categorized as functional(for example if the purpose of the system changes then the services delivered by it should reflect this change), non-functional(for example, the performance and reliability of the system), and technological (for example, the software and hardware aspects). In our system, changes due to volatility can lead to degradation in the quality of service of the service composition. Hence, the nature of change is non-functional.
c) Frequency: This dimension captures how often a particular change occurs (Andersson et al., 2009). The changes due to volatility occur frequently with respect to other distributed systems (Coulouris, 2012).

d) Anticipation: This dimension expresses whether a change can be predicted (Andersson et al., 2009). In our system, device and communication link failures, variation in the properties of communication such as bandwidth, and the creation and destruction of associations between software components resident on the devices can be foreseeable and hence planned for.

3) Mechanisms: This category of dimensions encapsulate the reaction of the system towards change (Andersson et al., 2009). In this category, each dimension's definition and the degree of each of those dimensions that our system exhibits are as follows:

a) Type: This dimension expresses if adaptation is related to the parameters of the system components or to the structure of the system (Andersson et al., 2009). The adaptations in our system are related to its structure- i.e. replacing a malfunctioning service in the composition with another equivalent service. However, our system is not designed to handle the situation where there is no equivalent service available. The question as to how to use an equivalence checker to decide if a service that is (say) 60 percent equivalent should be used as a replacement service is part of our future work.
b) Autonomy: This dimension is concerned with the degree of outside intervention during adaptation (Andersson et al., 2009). Adaptations are autonomous in our system since there is no outside intervention to effect adaptation.
c) Organization: This dimension identifies if the adaptation is done by a single component or distributed amongst several components (Andersson et al., 2009). In the case of our system, the adaptation is done by a single component.
d) Scope: This dimension captures whether adaptation is localized or involves the entire system (Andersson et al., 2009). The adaptation of our system is localized to replacing a single service when it malfunctions.
e) Duration: This dimension describes how long the adaptation lasts (Andersson et al., 2009). In our system, a malfunctioning service should be replaced in a short duration so that the system can continue to function without a long queue of faults to be serviced building up.
f) Timeliness: This dimension is concerned with whether the time period for performing self-adaptation can be guaranteed (Andersson et al., 2009). Our system makes a best effort to replace a malfunctioning service. In future work, we need to investigate if it is possible to guarantee that adaptation will take place before another change needs to be dealt with.
g) Triggering: This dimension captures whether the change that triggers an adaptation is associated with an event or a time slot (Andersson et al., 2009). In our system, adaptation is triggered by events that represent faults of the service composition.

4) Effects: The dimensions of adaptation in this category identify the impact of adaptation on the system (Andersson et al., 2009). We now explain the definition of each dimension and the degree of each of those dimensions that our system exhibits.

a) Criticality: This dimension captures the impact upon the system in case the self-adaptation fails (Andersson et al., 2009). The self-adaptation in our system is mission critical- if a malfunctioning service is not replaced, the service composition will fail.
b) Predictability: This dimension describes whether the consequences of adaptation can be predictable (Andersson et al., 2009). In our system's case, the adaptation is deterministic because we assume that only an equivalent service replaces a malfunctioning one and so the consequences of adaptation can be predicted. It is in this sense that the adaptation is deterministic.
c) Overhead: This dimension refers to the impact of system adaptation upon the quality of services of the system (Andersson et al., 2009). We postulate that our strategy of pre-caching equivalent service ensures that the overhead is low. Performance evaluation of models at runtime is part of our future work.
d) Resilience: This dimension is concerned with the persistence of service delivery that can justifiably be trusted when facing changes (Andersson et al., 2009). Again, we postulate that feedback control techniques are needed to guarantee resilience of the model at runtime- this is part of our future work.

We have analyzed our system with respect to the taxonomy of modeling dimensions of self-adaptive systems. The issues that have emerged are: i) How to deal with the situation where a replacement service is not entirely equivalent to a malfunctioning service? ii) How to guarantee timeliness of adaptation? iii) Performance evaluation of models at runtime, iv) Using feedback control techniques with a model at runtime to guarantee the resilience of response. In the next section, we tackle the performance evaluation of models at runtime. We leave the remaining issues as future work.

TABLE 6-1: Modelling dimensions for self-adaptive software systems(Andersson et al., 2009).

Dimensions	Degree	Definition
Goals- goals are objectives the system under consideration should achieve		
Evolution	Static to dynamic	Whether the goals can change within the lifetime of the system
Flexibility	Rigid, constrained, unconstrained	Whether the goals are flexible in the way they are expressed
Duration	Temporary to persistent	Validity of a goal through the system lifetime
Multiplicity	Single to multiple	How many goals are there?
Dependency	Independent to dependent (complimentary to conflicting)	How the goals are related to each other?
Change-change is the cause of adaptation		
Source	External (environmental), internal (application, middleware, infrastructure)	Where is the source of change?
Type	Functional, non-functional, technological	What is the nature of change?
Frequency	Rare to frequent	How often a particular change occurs?
Anticipation	Foreseen, foreseeable, unforeseen	Whether change can be predicted?
Mechanisms- what is the reaction of the system towards change		
Type	Parametric to structural	Whether adaptation is related to the parameters of the system components or to the structure of the system
Autonomy	Autonomous to assisted (system or human)	What is the degree of outside intervention during adaptation?
Organization	Centralized to decentralized	Whether the adaptation is done by a single component or distributed amongst several components
Scope	Local to global	Whether adaptation is localized or involves the entire system
Duration	Short, medium, long term	How long the adaptation lasts
Timeliness	Best effort to guaranteed	Whether the time-period for performing self-adaptation can be guaranteed
Triggering	Event-trigger to time trigger	Whether the change that triggers adaptation is associated with an event or a time slot
Effects- what is the impact of adaptation upon the system		
Criticality	Harmless, mission-critical, safety-critical	Impact upon the system in case the self-adaptation fails
Predictability	Non-deterministic to deterministic	Whether the consequences of adaptation can be predictable
Overhead	Insignificant to failure	The impact of system adaptation upon the quality of services of the system
Resilience	Resilient to vulnerable	The persistence of service delivery that can justifiably be trusted, when facing changes

6.3 A QUANTITATIVE PERFORMANCE EVALUATION OF THE RESPONSE TIMES OF OUR BIGRAPHICAL MODEL AT RUNTIME

As discussed in the previous section, one of the issues that has emerged out of the analysis of our system with respect to the taxonomy of modeling dimensions of self-adaptive systems is the need for a performance evaluation of our Bigraphical model at runtime. Our Bigraphical model is built on top of the BPL Tool's matching algorithm. However, this matching algorithm is known to be designed for correctness not efficiency (Elsborg, 2009). Moreover, the matching problem itself is NP-Complete (Birkedal et al., 2007). We wish to now quantify the effect of these on our system with realistic workloads.

To quantify the effects, we define response time as follows. If our model at runtime is looked upon as a black box where service composition faults are the inputs in the form of events and appropriate adaptation commands are the outputs, then we could define response time of our system as the time interval between inputting of events and outputting of adaptation commands.

The response time to process a particular event must be less than the time difference between this event and the next event. If this is not the case, then a well known queuing theory result tells us that a queue of requests to process events will build up with the average queuing delay growing without bound (Saltzer and Kaashoek, 2009).

To characterize the effect of the inefficiency of the matching algorithm of the BPL Tool and the intrinsic NP-Complete nature of the matching problem on our system, we have measured the response times of our system running on a laptop for realistic workloads. These workload events are generated by an app running on an Android device and are sent to the laptop through a TCP connection over an SSH tunnel.

We now discuss our experiments in detail. We first discuss the design of the test rig that we have used for all our experiments in section 6.3.1. Next, in section 6.3.2 we discuss the design of the experiments that we ran on the test rig. Then, in section 6.3.3 we describe the running of each set of our experiments and analyze the resulting data. In section 6.3.4 we discuss the experiments to find the cause of the exponential response times that we found in the previous section. Following this, in section 6.3.5,we discuss experiments that measure the effect of the workload events on the available time. Finally, in section 6.3.6, we summarize and discuss our experimental results.

6.3.1 DESIGN OF THE TEST RIG

We have designed a test rig to conduct our experiments to evaluate the Bigraphical model at runtime. The architecture of the test rig consists of an Android phone running our simulation script in Java and a laptop running our SML code of the Bigraphical model at runtime.

On the Android phone, we simulate the generation of the following events as discussed in the previous chapters: (a) One of the services participating in the composition develops a fault, (b) The user moves from one ambient to another. The Android phone sends these events to the laptop through a TCP connection over an SSH tunnel. We run the simulation script on an Android device because in our scenario, events that are triggered by a system outside our system boundary are being generated on the mobile device for our model at runtime to respond to- and we simulate this through the Shopping Mall mobility model (discussed later) running on the Android device.

Our SML implementation of the Bigraphical model running on the laptop then responds to an event by sending back an appropriate command to the Android phone through the TCP connection over an SSH tunnel.

We instrument the code at both the SML end and the Android end to measure response times, time difference between events etc.

The hardware and software specifications of our test rig are as follows:

(i) To run the Android app that generates the workload of events, we have used a Samsung Galaxy S 5.0 YPG70 device. This device runs the CyanogenMod open-source operating system version 10.1 based on the Android mobile platform. Its Kernel version is 3.0.75Mercurious_reborn_rc1. This device has an ARM v7 processor rev2 (v71) and has 512 MiB of memory.

(ii) The BPL Tool code is targeted for compilation at the SML/NJ compiler (Standard ML of New Jersey, version 110.69). The laptop that runs our code using this BPL Tool is a Dell machine with 512 MiB of memory and a 1.6 GHz Celeron processor. This laptop's operating system is Ubuntu 10.04 Lucid Lynx with kernel version 2.6.32-52 generic. We have used a laptop rather than an Android machine because to the best of our knowledge, the SML/NJ version 110.69 has not been ported to run on the Android Software Stack.

(iii) The Android app that generates the workload of events implements the Shopping Mall Mobility model (Galati et al., 2013). For this implementation, we have downloaded and used the following Java code and libraries:

a. Recipe 6.7 Creating a Socket Server, and Recipe 10.5 Saving files to external storage both downloaded from Lee's website (Lee, 2013).
b. Code for the Room class from the text book of Barnes and Kolling (Barnes and Kolling, 2013).
c. The JGraphT library version 0.9.0 downloaded from the website of JGraphT (JGraphT,2014).
d. The Commons Mathematics Library version 3.2 from Apache (Apache, 2014).

We now discuss how our architecture is informed by the organization of an actual deployment. In such a deployment an actual user would be moving around in a shopping mall with a service composition running on her mobile phone. Our Bigraphical model at runtime would be running on this same device making sure that the services that develop a fault are appropriately replaced. Also, the user's location in the mall is constantly updated in the model at runtime as the user moves around.

Although, we envisage that our Bigraphical model at runtime runs on a mobile device, for our experiments we are interested in the shape of the curve for average response times as a function of the number of nodes in a Bigraph. This shape in the form of a mathematical equation will be independent of the machine on which the Bigraph model is running. Therefore, running the Bigraphical model at runtime on a laptop instead of a mobile device (because the SML/NJ compiler has not been ported to a mobile device as discussed earlier) will make no difference to the conclusions about the shape of the curve that we will draw from our experiments.

Similarly, notwithstanding a TCP connection over an SSH tunnel between our Bigraphical model at runtime running on a laptop and the event generator on the Android device, because we measure our response times on the laptop, our results are not affected by the network delays inherent in TCP connections.

6.3.2 DESIGN OF THE EXPERIMENTS

We now discuss the design of our experiments. Firstly, we explain the Shopping Mall mobility model that we used to write an Android app to generate the realistic workloads on our Bigraphical model at run time. Then, we discuss the semantics that we followed to simulate the generation of events by our Android app. and define the available time within which our functions written in SML and running on the laptop must respond. Following this, we describe the three sets of experiments that we conducted. Finally, we discuss the statistical techniques that we used to analyse our data.

6.3.2.1 THE SHOPPING MALL MOBILITY MODEL

For simulating the event generation on the Android phone, we have implemented the Shopping Mall Mobility Model as discussed in Galati, Djemame and Greenhalgh's paper (Galati et al., 2013) . Following are the salient points of the implemented model:

a) The overall time spent by a user in a Shopping Mall is characterized by the random sampling of a Weibull cumulative distribution function (shape=0.935, scale=2.579e+03).

b) The time spent by a user in a particular shop is characterized by the random sampling of another Weibull cumulative distribution function (shape=1.002, scale=3.059e+02).

c) The topology of the shopping mall is a graph. Each node is a Mall intersection (i.e. either the common area where the user needs to change direction, or the shops). The user randomly selects a target shop to go to from the shop in which they are currently located. This selection of the target shop depends on its attraction level. However, in the Galati et al. paper (Galati et al., 2013) all shops are set to the same attraction level of 1 and we follow the same strategy. The route from the current shop to the destination shop is found by using Dijkstra's shortest path algorithm.

d) The user moves though each intermediate node en-route to the target shop by following these rules:

(i) The speed of the user between the midpoints of two consecutive nodes is randomly selected from the interval [1.15m/s – 1.65m/s]. The distance between the midpoints of the nodes is assumed to be 5 meters. This distance approximately models the distances between shops in the Churchill Square Shopping Mall, Brighton.

(ii) At the mid-point of each intermediate node, the user pauses for a time randomly selected from the interval [0s-2s] perhaps to change direction or speed.

e) Once the user reaches the target shop, she spends time in the shop as described in (b).

f) Steps (c) to (e) are repeated until the user exhausts the time allocated to them in step (a).

We have used six random number generators with six different seeds in our Shopping mall mobility model simulation. To maintain consistency across experiments, these same seeds have been used again and again for all experiments.

We now describe how the time spent by the user as per the Shopping Mall Mobility model maps onto the topology of the shopping mall of our scenario.

Across all our experiments, the three ambients `ShoppingMall`, `loc1` and `loc3` always play the following roles:

- `ShoppingMall` represents a foyer connecting the east and the west wings.
- `loc1` represents a corridor in the west wing along which the shops are located.
- `loc3` represents a corridor in the east wing along which the shops are located.

Thus, the above three ambients are the way-points in our model whereas all the other ambients represent shops (See Figure 6-1).

Now there are two kinds of pauses of a shopper in the Shopping Mall mobility model that we have discussed above.

The first pause is for a time randomly selected from the interval [0s-2s] which we assume occurs at the *midpoint* of the three ambients the `ShoppingMall`, `loc1` and `loc3` (See Figure 6-2). The user pauses *only* for this time at these way-points perhaps to change direction or speed (Galati et al., 2013).

The other pause occurs *only* in a shop (See Figure 6-3) and is characterized by the random sampling of a Weibull cumulative distribution function (shape=1.002, scale=3.059e+02).

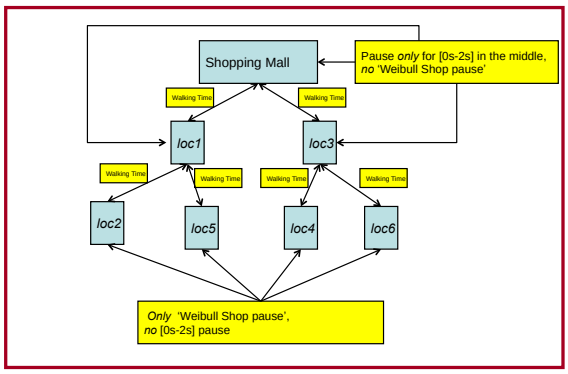

FIGURE 6-1: TWO CLASSES OF AMBIENTS-IN ONE CLASS, THE SHOPPER PAUSES AT THE MID-POINT BETWEEN [0S-2S] WHEREAS IN THE OTHER CLASS, THE SHOPPER PAUSES FOR A TIME CHARACTERIZED BY THE RANDOM SAMPLING OF A WEIBULL CUMULATIVE DISTRIBUTION FUNCTION (SHAPE=1.002, SCALE=3.059E+02).

Notice that the entire time spent in any one of the three way-points `ShoppingMall`, `loc1` and `loc3` is *not* between [0s-2s]. Consider for example `loc1` in the Figure 6-2. The total time spent in `loc1` will be the *sum* of the first walking time from the left-hand boundary of `loc1` up-to the midpoint, the pause between [0s-2s] at the midpoint and finally the second walking time from the midpoint of `loc1` to the right-hand boundary of `loc1`. We distinguish between the first and

the second walking times as the two could have different values. This is because the speed between the midpoints of any two ambients is randomly selected from the interval [1.15m/s-1.65m/s] as discussed earlier. Moreover, the distance between the midpoints of any two consecutive ambients is assumed to be 5m. Hence the time spent walking between the midpoint of an ambient preceding `loc1` and the midpoint of `loc1` itself could be different from the time spent walking between the midpoint of `loc1` and the midpoint of the subsequent ambient. Similar arguments hold for the entire time spent in the ambients `ShoppingMall` or `loc3`.

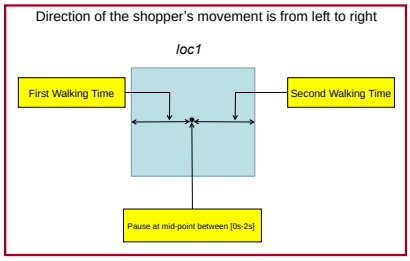

FIGURE 6-2: THE TOTAL TIME SPENT IN THE WAY-POINT `loc1`.

On the other hand, the total time spent in a particular shop can be calculated as follows: Consider `loc2` in Figure 6-3. The entire time spent in `loc2` will be the sum of the walking time from the left-hand boundary of `loc2` up-to the midpoint, the pause at the midpoint characterized by the random sampling of a Weibull cumulative distribution function (shape=1.002, scale=3.059e+02) (this pause models the fact that the user looks around the shop but we assume that she returns to the midpoint to go out of the shop), and finally the walking time from the midpoint of `loc2` to the left-hand boundary of `loc2`.

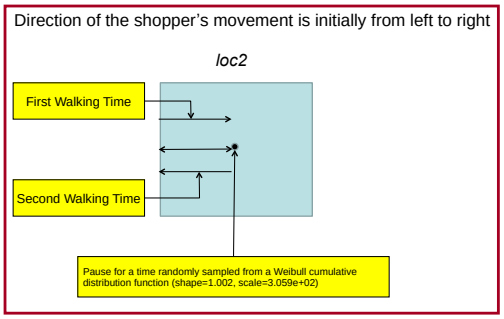

FIGURE 6-3: THE TOTAL TIME SPENT IN THE SHOP `loc2`.

6.3.2.2 A DESCRIPTION OF THE THREE SETS OF EXPERIMENTS

Using the template discussed in the previous section, we have conducted the following 3 sets of experiments. Each set is different from the other set in terms of some parameter. Our goal in varying some parameter is to see if the response times of our SML functions for different variations in experiments can be characterized using regression analysis.

(a) The set of experiments where for each experiment, we increase the number of location nodes in our SML Bigraphical model of the previous experiment by one. Then, for the resulting Bigraphical model on the laptop, we run the simulation by generating events on the Android machine and sending them to the model. As discussed earlier, these events are based on the Shopping Mall Mobility model. We repeat these experiments each time increasing the number of location nodes by one till the SML system on the laptop keels over (i.e. stops responding to the event sent to it). At this point, the CPU utilization for the process representing our system is observed to be 100% and the system is observed to be thrashing.

(b) The set of experiments where one of the three services participating in the service composition never disappears and is available across the shopping mall. For each experiment, we increase the number of location nodes in our SML Bigraphical model of the previous experiment by one. Then, for the resulting Bigraphical model on the laptop, we run the simulation by generating events on the Android machine and sending them to the model. As discussed earlier, these events are based on the Shopping Mall Mobility model. We repeat these experiments each time increasing the number of location nodes by one till the SML system on the laptop keels over (i.e. stops responding to the event sent to it). At this point, the CPU utilization for the process representing our system is observed to be 100% and the system is observed to be thrashing.

(c) The set of experiments where for each experiment, we increase the number of services participating in the composition in the previous experiment by one keeping the number of locations constant. Then, for the resulting SML Bigraphical model on the laptop, we run the simulation by generating events on the Android machine and sending them to the model. As discussed earlier, these events are based on the Shopping Mall Mobility model. We repeat these experiments each time increasing the number of services participating in the composition by one till the SML system on the laptop keels over (i.e. stops responding to the event sent to it). At this point, the CPU utilization for the process representing our system is observed to be 100% and the system is observed to be thrashing.

6.3.2.3 THE SEMANTICS OF THE GENERATION OF WORKLOAD EVENTS

As discussed earlier in the thesis, we have written two functions to encapsulate the adaptation logic and simulate test runs: `newChangeAmbientScript` function and the `SCAFaultScript` function. A user moving from one ambient to another is an event occurring at the WORLD layer in our scenario and is mapped to the function `newChangeAmbientScript`. Similarly, events at the SCA layer trigger the function `SCAFaultScript`. In the following discussion, we will use the abbreviation 'n' for a call to the function `newChangeAmbientScript` and 's' for a call to `SCAFaultScript`.

We define the workload for our Bigraphical model at runtime as the sequence of events that occurs as the user moves around in the shopping mall.

The *available time* for a given function to respond is the time difference between the occurrence of the event that triggered the function and the next event.

Now, the distribution of the times of the occurrences of the events 'n' and 's' are not based on any realistic model in our simulation. Instead we will consider two scenarios- a favorable and an unfavorable scenario in the discussion in section 6.3.5.

Note too that we are measuring the response time by timing the running of a function (`newChangeAmbientScript` or `SCAFaultScript`) on the laptop. This is because our research proposal envisages that both the events and their corresponding commands are generated on the same mobile phone that is running our model at runtime.

Thus, we do not calculate the response time as the time between an event being sent from the Android machine and the command being received back from the laptop. This architecture where events are generated on the Android machine and then sent to the laptop through a TCP connection over an SSH tunnel and the laptop responds with an appropriate command back to the Android machine is meant only for conducting tests.

Following are some additional semantics of our simulation code running on the Android machine to generate the events that are required to trigger the two functions `newChangeAmbientScript` and `SCAFaultScript`:

- We assume that when a device malfunctions at a given location, the replacement device appears in this same location. This information is provided by a system outside our system boundary. The Android device then sends the id of the new device and its location as part of the generation of the event corresponding to this malfunction to the laptop for its response.

- We assume that a system outside our system boundary has already initialized our model and so all the backup devices are cached in all the locations before our simulation starts. As discussed in the previous chapter, the backup devices are numbered according to the following scheme: Each device id is a decimal number. The number on the left hand side of the decimal point represents the service number that the device offers. The number on the right hand side of the decimal point represents the ordinal number of the device in our model offering this particular service. For example a device with the id "2.5" represents the fifth device in our model that offers service number two. Initially, our model looks as shown in Figure 6-4. In our model, the device id "0.0" always represents the user's mobile device.

The implication of the above two points is that when a replacement device appears (is cached) in our model, its ordinal number is one more than the highest ordinal number of a device that is offering the same service. For example, in Figure 6-4, let us suppose that the device with id "1.3" disappears from `loc2` (i.e. location with the id `i2` in Figure 6-4). Then the replacement device that will appear in `loc2` will have the id "1.6" since the highest ordinal number of a device offering service 1 anywhere in the model as shown in Figure 6-4 is "1.5".

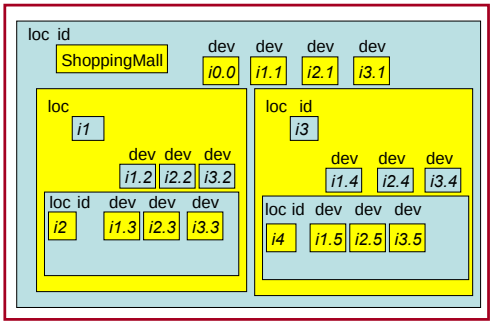

FIGURE 6-4:THE INITIAL STATE OF THE WORLD BIGRAPH FOR THE FIRST EXPERIMENT.

6.3.2.4 THE STATISTICAL ANALYSIS USED FOR THE EXPERIMENTS

For each individual experiment in the three sets of experiments discussed above, we calculate the mean response time with a 95% confidence interval for the function `newChangeAmbientScript`. Thus, for a given set of experiments, we get a set of mean response times each with a 95% confidence interval. We then run a regression analysis on this set of mean response times to generate a best fitting curve that characterizes the increase in the mean response time as the number of nodes in the Bigraph model increase.

We then repeat this process for the same set of experiments but this time for the function `SCAFaultScript` and get another best fitting curve from the regression analysis. This curve

characterizes the mean response time's increase for the function `SCAFaultScript` as the size of the Bigraph increases for this particular set of experiments.

We have used MATLAB scripts to conduct the statistical analysis of our data.

For calculating the 95% confidence intervals of the mean response time in our MATLAB scripts, we have used the following mathematical expressions:

$(\mu - (1.96 * s / \sqrt{(n-1)}) , (\mu + (1.96 * s / \sqrt{(n-1)})$

where n is the sample size, μ is the mean and s is the standard deviation. These expressions are valid both for normal and non-normal populations with unknown variance and a large sample size (n >= 30) (Crawshaw and Chambers, 2001).

We now give an example MATLAB script (Gilat, 2009) to show how we have conducted our regression analysis with MATLAB:

- Create vector t and w with coordinates of the data points:
  t = 0 : 0.5 : 5;
  w = [6 4.83 3.7 3.15 2.41 1.83 1.49 1.21 0.96 0.73 0.64]
- Use the *polyfit* function of MATLAB with *t* and log(*w)* and pass 1 for a polynomial of degree 1. Curve fitting with polynomials is done in MATLAB with the *polyfit* function, which uses the least squares method. MATLAB's *log* function calculates the natural logarithm of its argument.
  p = polyfit (t, log(w), 1) ;
  m = p (1)
- Determine the coefficient *b*.
  b = exp (p(2))
- Create a vector *tm* to be used for plotting the polynomial.
  tm = 0 : 0.1 : 5;
- Calculate the function value at each element of *tm*. Here, the *exp* function of MATLAB calculates e^x for any argument *x*.
  *wm = b * exp (m * tm);*
- Plot the data points and the function using the MATLAB *plot* function.
  plot(t, w, 'o' ,tm, wm)

When the above script is executed, MATLAB's Command Window displays the values of the constants *m* and *w* (Gilat, 2009).

m =

-0.4580

b =

5.9889

The following plot (Figure 6-5) is generated with the MATLAB script discussed above:

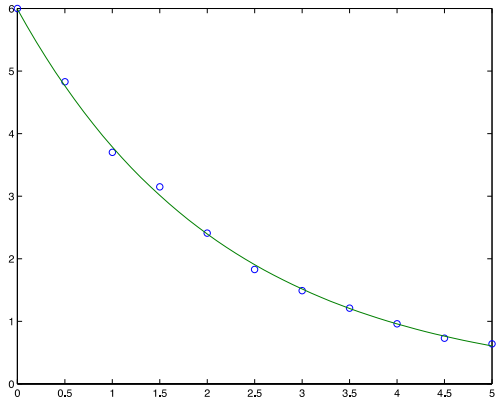

FIGURE 6-5: AN EXAMPLE REGRESSION CURVE.

6.3.3 RUNNING OF THE EXPERIMENTS AND ANALYSING THE DATA

We now discuss the running of each of the three sets of experiments in detail and analyze the data that has been generated.

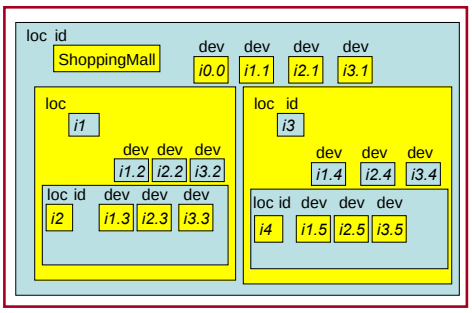

FIGURE 6-6: THE INITIAL STATE OF THE WORLD BIGRAPH FOR THE FIRST EXPERIMENT.

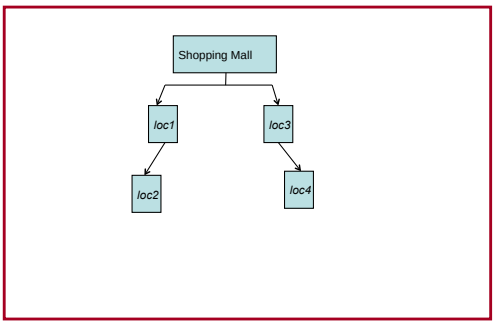

FIGURE 6-7:AN ABSTRACTED OUT TREE STRUCTURE OF THE LOCATION MODEL FOR EXPERIMENT 1.

(a) *Experiment where we successively increase the number of location nodes by one and run the simulation based on the Shopping mall Mobility model for each of the resulting topology of the shopping mall*: We start with the basic topology of two shops represented by `loc2` and `loc4` in the two wings of the Shopping mall as shown in Figure 6-6. In Figure 6-7 we abstract out the tree structure of the locations. For this topology, we run our simulation using the template discussed in the previous section. In the next experiment, we add `loc5` to this basic topology. Notice that the new location `loc5` also has been initialized like other locations with three devices that are the backup devices for the three services participating in the composition. As discussed in the previous chapter, each device is itself a Bigraph node that is composed of five smaller Bigraph nodes. Thus, the addition of three devices means an addition of fifteen more Bigraph nodes. Moreover, the location node itself is composed of three smaller Bigraph nodes. As a result, adding a single location in effect adds eighteen more nodes. We again run our simulation for this larger Bigraphical model.

Continuing in this manner, we keep on increasing the number of location nodes by one and then running the simulation for that particular topology of the shopping mall. The system keels over (i.e. stops responding) when we add `loc11`. Thus, our data corresponds to a total of seven experiments. The first experiment corresponds to a simulation that runs on a shopping mall model with locations up to `loc4` whereas the last experiment corresponds to a simulation that runs on a shopping mall model with locations up to `loc10`.

Figures from Figure 6-8 to Figure 6-13 show the abstracted out tree structure of the location model used for the six experiments following the first one:

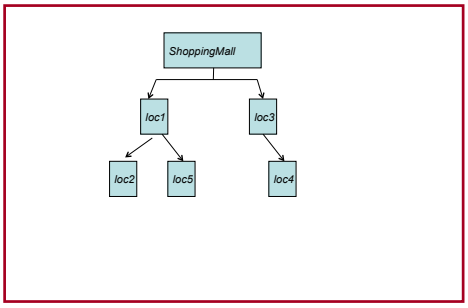

FIGURE 6-8: AN ABSTRACTED OUT TREE STRUCTURE OF THE LOCATION MODEL FOR EXPERIMENT 2.

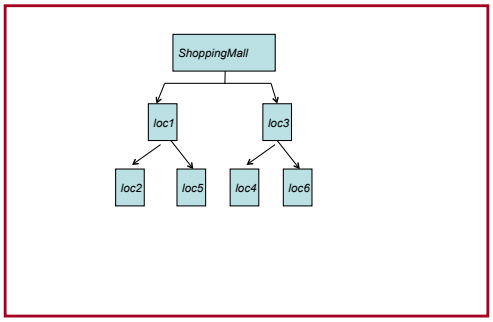

FIGURE 6-9: AN ABSTRACTED OUT TREE STRUCTURE OF THE LOCATION MODEL FOR EXPERIMENT 3.

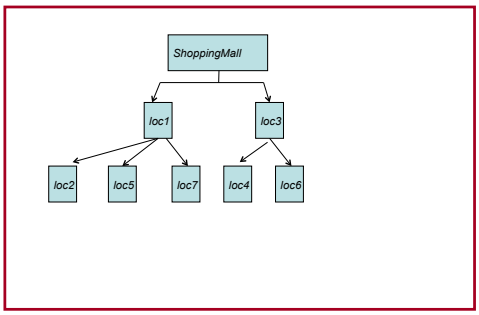

FIGURE 6-10: AN ABSTRACTED OUT TREE STRUCTURE OF THE LOCATION MODEL FOR EXPERIMENT 4.

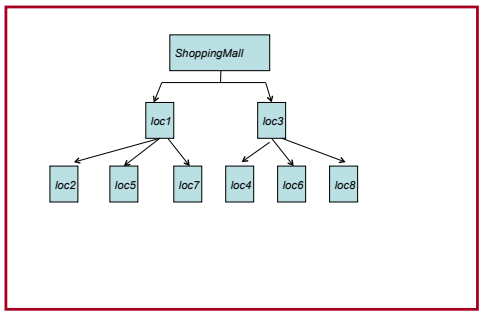

FIGURE 6-11: AN ABSTRACTED OUT TREE STRUCTURE OF THE LOCATION MODEL FOR EXPERIMENT 5.

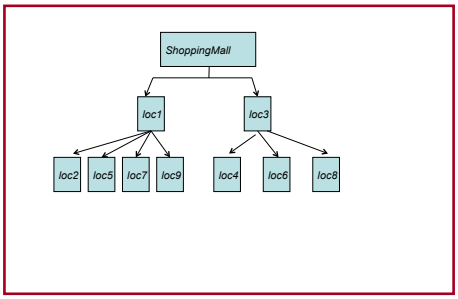

FIGURE 6-12: AN ABSTRACTED OUT TREE STRUCTURE OF THE LOCATION MODEL FOR EXPERIMENT 6.

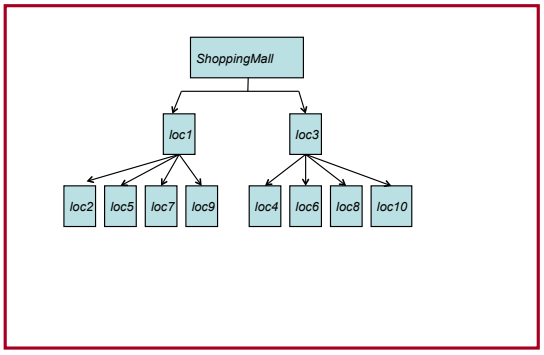

FIGURE 6-13: AN ABSTRACTED OUT TREE STRUCTURE OF THE LOCATION MODEL FOR EXPERIMENT 7.

We summarize the data of this set of experiments with the three figures from Figure 6-14 to Figure 6-16.

Figure 6-14 is a representative figure that shows the data corresponding to experiment number three with the shopping mall having locations up to `loc6`. Each bar in the graph represents the response time as the user moves around the shopping mall as per the Shopping Mall Mobility Model.

The second figure, Figure 6-15 shows seven mean response times with a 95% confidence interval for calls to the `newChangeAmbientScript` function (abbreviated as 'n' in the figure) for each of the seven experiments discussed above.

This second figure also shows an exponential best fitting curve for the seven mean response times. This curve has been generated through a regression analysis done using a MATLAB script. The exponential function is given by the expression

$y = be^{mx}$

The values of the constants b and m calculated by the MATLAB script are:

b = 2.853e+03

m = 0.3044

Similar to the second figure, we have a third figure, Figure 6-16 that shows seven mean response times with a 95% confidence interval for calls to the `SCAFaultScript` function (abbreviated as 's' in the figure) for each of the seven experiments discussed above. This third figure also shows an exponential best fitting curve for the seven mean time gaps. This curve too has been generated through a regression analysis done using a MATLAB script. The exponential function is given by the expression

$y = be^{mx}$

The values of the constants *b* and *m* calculated by the MATLAB script are:

b = 6.9485e+03, m = 0.3320

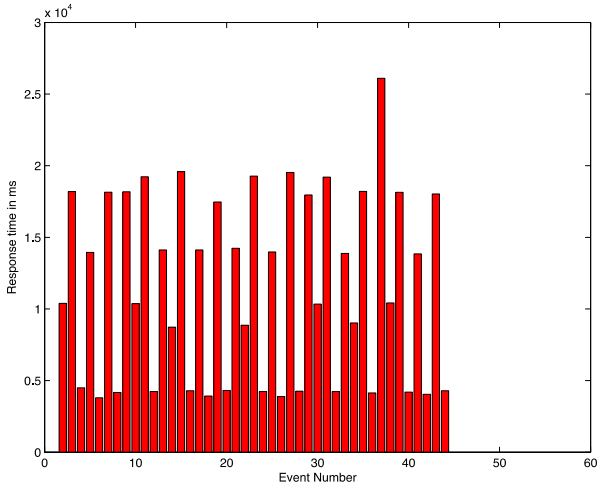

FIGURE 6-14: THE RESPONSE TIMES IN MILLISECONDS FOR EXPERIMENT NUMBER 3.

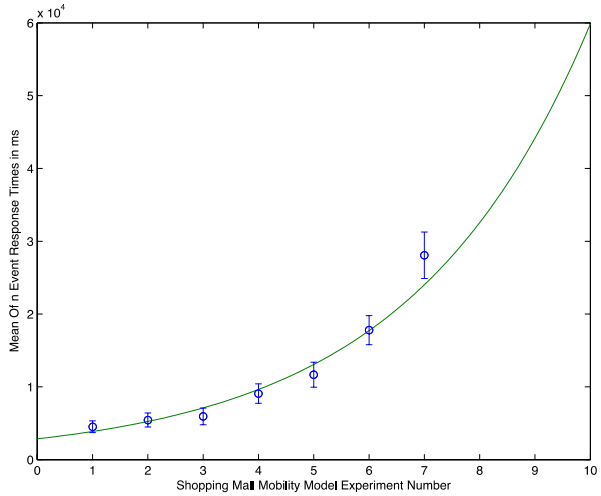

FIGURE 6-15: REGRESSION CURVE OF THE SEVEN MEAN RESPONSE TIMES FOR SEVEN EXPERIMENTS WITH A 95% CONFIDENCE INTERVAL FOR CALLS TO FUNCTION ‘n’.

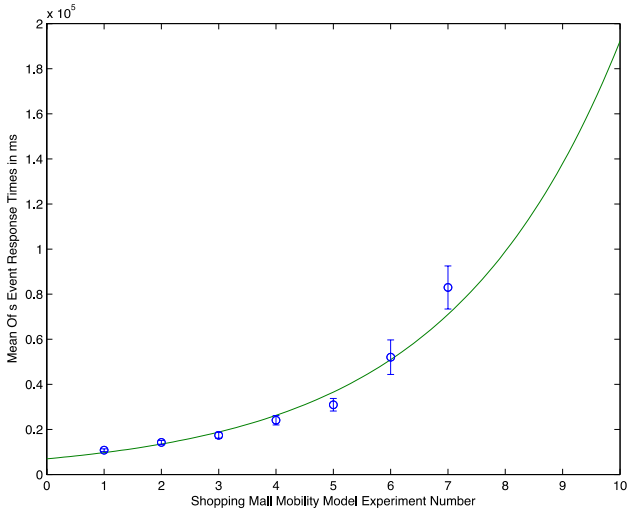

FIGURE 6-16:REGRESSION CURVE OF THE SEVEN MEAN RESPONSE TIMES FOR SEVEN EXPERIMENTS WITH A 95% CONFIDENCE INTERVAL FOR CALLS TO FUNCTION 's'.

Thus, in this experiment, the response times increase exponentially for both 'n' and 's' events as the number of nodes is increased.

(b) *Experiment where one of the three services participating in the service composition never disappears and is available across the shopping mall :* In this experiment, we do not cache backup devices for service1 in any location except the node called `ShoppingMall` which represents the common area of the mall. We assume that service 1 is a 'strong' service available across the shopping mall at all locations. Thus, it is only for services 2 and 3 that we have cached those devices that offer these services at all locations in our model. Once again, for the first experiment we start with the basic topology of two shops represented by loc2 and loc4 in the two wings of the Shopping mall as shown in Figure 6-6 and Figure 6-7. For this topology, we run our simulation using the template discussed in the previous section. In the next experiment, we add `loc5` to this basic topology. Notice that this time, the new location `loc5` also has been initialized like other locations with only two devices that are the backup devices for the two out of three services participating in the composition. As discussed in the previous chapter, each device is itself a Bigraph node that is composed of five smaller Bigraph nodes. Thus, the addition of two devices means an addition of ten more Bigraph nodes. Moreover, the location node itself is composed of three smaller Bigraph nodes. As a result, adding a single location in effect adds thirteen more nodes. We again run our simulation for this larger Bigraphical model.

Continuing in this manner, we keep on increasing the number of location nodes by one and then running the simulation for that particular topology of the shopping mall. The system keels over (i.e. stops responding) when we add `loc13`. Thus, this time our data corresponds to a total of

nine experiments. The first experiment corresponds to a simulation that runs on a shopping mall model with locations up to `loc4` whereas the last experiment corresponds to a simulation that runs on a shopping mall model with locations up to `loc12`.

Figures from Figure 6-8 to Figure 6-13 show the abstracted out tree structure of the location model used for the six experiments following the first one. Also Figure 6-17, Figure 6-18, and Figure 6-19 show the tree structure for the location model when we add `loc11`, `loc12` and `loc13` respectively:

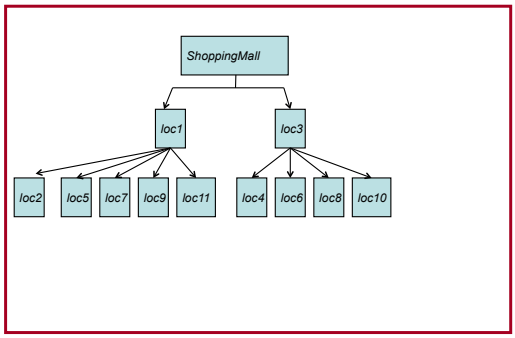

FIGURE 6-17: AN ABSTRACTED OUT TREE STRUCTURE OF THE LOCATION MODEL FOR EXPERIMENT 8.

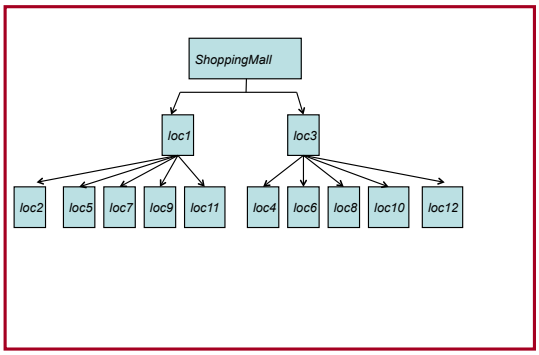

FIGURE 6-18: AN ABSTRACTED OUT TREE STRUCTURE OF THE LOCATION MODEL FOR EXPERIMENT 9.

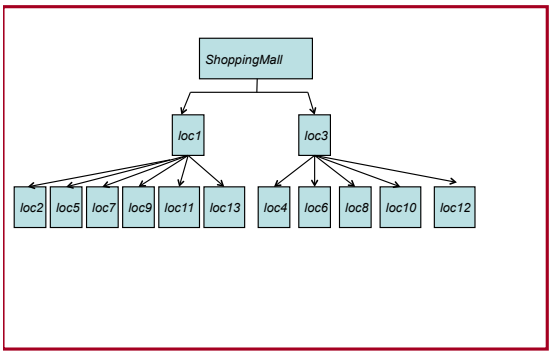

FIGURE 6-19: AN ABSTRACTED OUT TREE STRUCTURE OF THE LOCATION MODEL FOR EXPERIMENT 10 WHERE OUR SYTEM KEELS OFF.

We summarize the data of this set of experiments with the three figures from Figure 6-20 to Figure 6-22.

Similar to the previous set of experiments, the first figure, Figure 6-20 is a representative figure that shows the data corresponding to experiment number three with the shopping mall having locations up to `loc6`. Each bar in the graph represents the available time as the user moves around the shopping mall as per the Shopping Mall Mobility Model.

The second figure, Figure 6-21 shows nine mean response times with a 95% confidence interval for calls to the `newChangeAmbientScript` function (abbreviated as 'n' in the figure) for each of the nine experiments discussed above.

This second figure also shows an exponential best fitting curve for the nine mean response times. This curve has been generated through a regression analysis done using a MATLAB script. The exponential function is given by the expression

$y = be^{mx}$

The values of the constants *b* and *m* calculated by the MATLAB script are:

b = 1.8578e+03

m = 0.2674

Similar to the second figure, we have a third figure, Figure 6-22 that shows nine mean response times with a 95% confidence interval for calls to the `SCAFaultScript` function (abbreviated as 's' in the figure) for each of the nine experiments discussed above. This third figure also shows

an exponential best fitting curve for the nine mean response times. The exponential function is given by the expression

$y = be^{mx}$

The values of the constants *b* and *m* calculated by the MATLAB script are:

b = 4.4814e+03

m = 0.2899

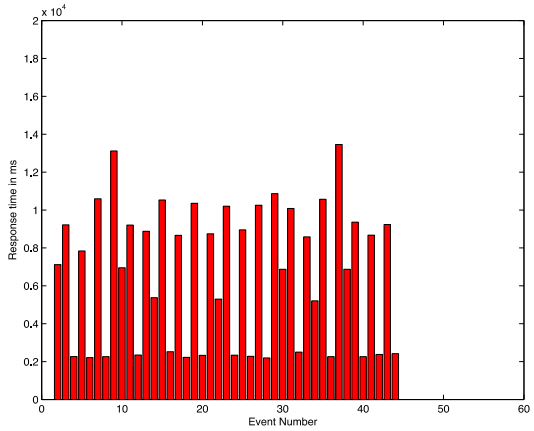

FIGURE 6-20:THE RESPONSE TIMES AND IN MILLISECONDS FOR EXPERIMENT NUMBER 3.

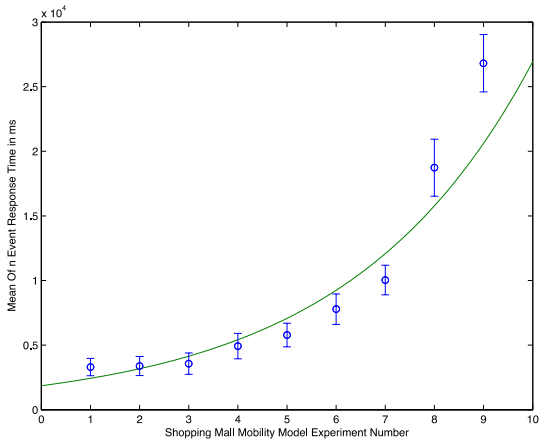

FIGURE 6-21:REGRESSION CURVE OF THE NINE MEAN RESPONSE TIMES FOR NINE EXPERIMENTS WITH A 95% CONFIDENCE INTERVAL FOR CALLS TO FUNCTION 'n'.

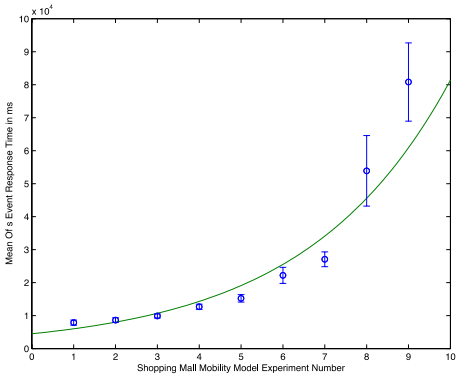

FIGURE 6-22:REGRESSION CURVE OF THE NINE MEAN RESPONSE TIMES FOR NINE EXPERIMENTS WITH A 95% CONFIDENCE INTERVAL FOR CALLS TO FUNCTION 's'.

Thus, in this experiment too, the response times increase exponentially for both 'n' and 's' events as the number of nodes is increased.

(c) *Experiment where we successively increase the number of services participating in the composition keeping the number of locations constant*: For the first experiment, we start with the basic topology of two shops represented by `loc2` and `loc4` in the two wings of the Shopping mall as shown in Figure 6-6 and Figure 6-7 and with three services participating in the composition. For this topology, we run our simulation using the template discussed in the previous section. In the next experiment, we add a fourth service to the composition. Because we cache backup devices for a service in all locations, we initialize the WORLD Bigraph such that there are five additional devices offering this fourth service in each of the locations namely `ShoppingMall`, `loc1`, `loc2`, `loc3`, and `loc4`. Since each device is itself a Bigraph that is composed of five smaller Bigraph nodes, adding five additional cached devices results in adding twenty-five bigraph nodes to the new topology. Moreover, the service node itself is composed of four smaller Bigraph nodes. Thus when we increase the number of services participating in the composition by one, a total of twenty-nine additional nodes are added to the new topology. We again run our simulation for this larger Bigraphical model.

Continuing in this manner, we keep on increasing the number of services participating in the composition by one and also caching backup devices that offer this service in all the five locations. The system keels over (i.e. stops responding) when we add a sixth service and its backup devices. Thus, for this set of experiments our data corresponds to three experiments. The first experiment corresponds to a simulation that runs on a shopping mall model with three services participating in the composition whereas the last experiment corresponds to a simulation that runs on a shopping mall model with five services participating in the composition. Note that for this set of experiments, the number of location nodes remains the same.

Figures from Figure 6-23 to Figure 6-26 show the structure of the service composition starting with three services and increasing the number of services by one.

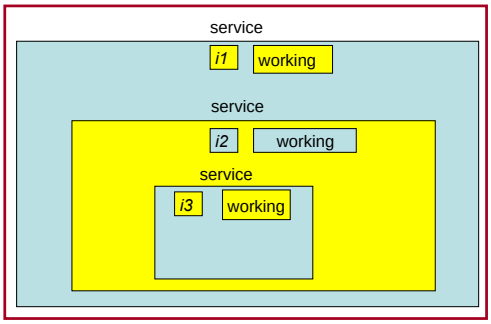

FIGURE 6-23: 3 SERVICES IN THE COMPOSITION.

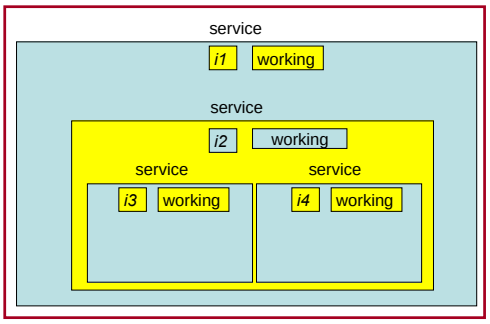

FIGURE 6-24: 4 SERVICES IN THE COMPOSITION.

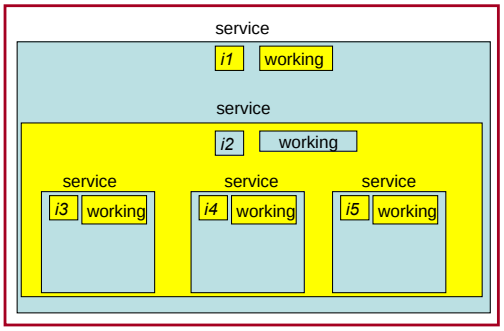

FIGURE 6-25: 5 SERVICES IN THE COMPOSITION.

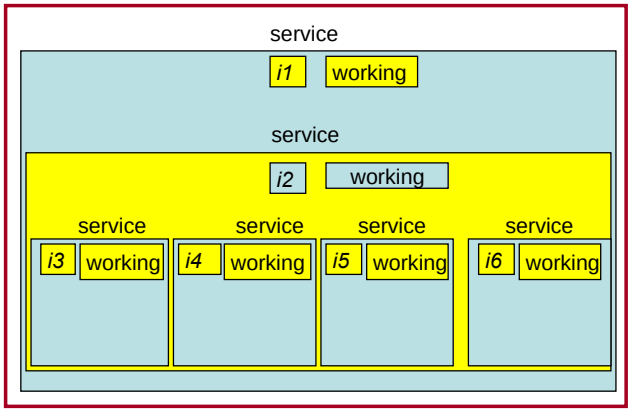

FIGURE 6-26: 6 SERVICES IN THE COMPOSITION.

We summarize the data of this set of experiments with the three figures from Figure 6-27 through to Figure 6-29.

The first figure, Figure 6-27 is a representative figure that shows the data corresponding to experiment number three with five services participating in the composition. As before, each bar in the graph represents the response time as the user moves around the shopping mall as per the Shopping Mall Mobility Model.

The second figure, Figure 6-28 shows three mean response times with a 95% confidence interval for calls to the `newChangeAmbientScript` function (abbreviated as 'n' in the figure) for each of the three experiments discussed above.

This second figure also shows an exponential best fitting curve for the three mean response times. This curve has been generated through a regression analysis done using a MATLAB script. The exponential function is given by the expression

$y = be^{mx}$

The values of the constants b and m calculated by the MATLAB script are:

$b = 1.3913e+03, m = 1.1695$

Similar to the second figure, we have a third figure, Figure 6-29 that shows three mean response times with a 95% confidence interval for calls to the `SCAFaultScript` function (abbreviated as 's' in the figure) for each of the three experiments discussed above. This third figure also shows an exponential best fitting curve for the three mean response times. The exponential function is given by the expression

$y = be^{mx}$

The values of the constants *b* and *m* calculated by the MATLAB script are:

b = 3.6622e+03

m = 1.0691

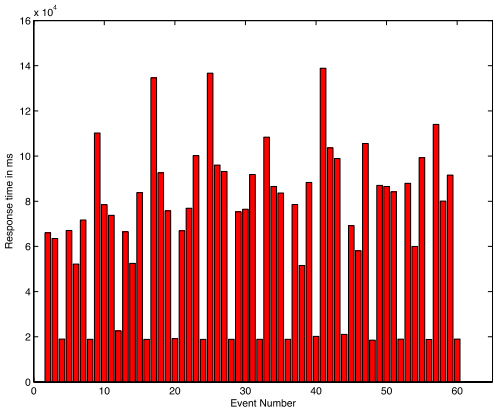

FIGURE 6-27: THE RESPONSE TIMES IN MILLISECONDS FOR EXPERIMENT NUMBER 2.

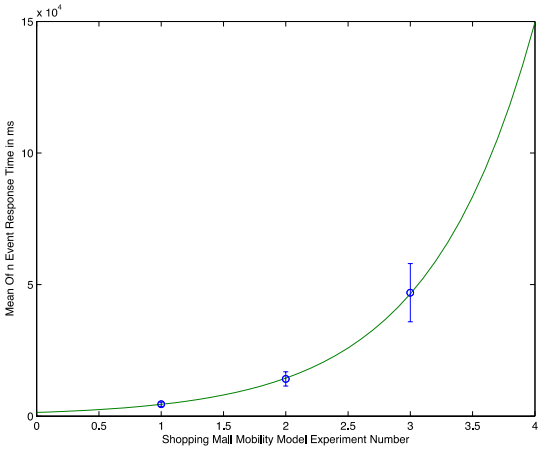

FIGURE 6-28:REGRESSION CURVE OF THE THREE MEAN RESPONSE TIMES FOR THREE EXPERIMENTS WITH A 95% CONFIDENCE INTERVAL FOR CALLS TO FUNCTION ‘n’.

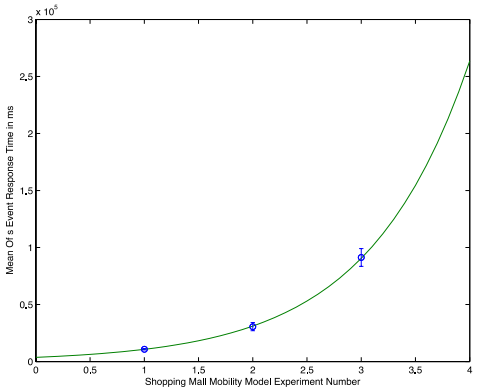

FIGURE 6-29:REGRESSION CURVE OF THE THREE MEAN RESPONSE TIMES FOR THREE EXPERIMENTS WITH A 95% CONFIDENCE INTERVAL FOR CALLS TO FUNCTION 's'.

Similar to the previous two experiments, in this experiment too the response times increase exponentially for both 'n' and 's' events as the number of nodes is increased.

6.3.4 CAUSE OF THE EXPONENTIAL INCREASE IN RESPONSE TIMES: A NAÏVE HANDLING OF THE DECOMPOSITION OF THE PRIME PRODUCT CHILDREN OF A NODE BY THE MATCHING ALGORITHM OF BPL TOOL (ITU, 2011),(BIRKEDAL ET AL., 2007)

The developers of the BPL Tool acknowledge that their implementation of matching is not very fast (ITU, 2011). This is because the matching depends on the normal form lemmas of Bigraphs (Birkedal et al., 2007). These lemmas express how larger Bigraphs can be decomposed into smaller ones. The matching proceeds by induction on these decomposed Bigraphs. In particular the normal forms cannot distinguish between prime products (the operation of placing Bigraphs side-by-side under a common parent node-See Chapter 2) for example $F_0 \mid F_1$ and $F_1 \mid F_0$. As a result when matching children of a node, if the children are combined in a prime product, the BPL Tool generates all the possible permutations of that prime product. Each of this permutation represents a separate decomposition. The matching implementation of the BPL Tool then explores all of these decompositions.

Consider for example Figure 6-13 which we repeat in Figure 6-30. The children of `loc1` are all constructed by using the prime product to place them side-by-side within `loc1`. To match the children of `loc1`, we need to first decompose it into its constituent children. Seven possible decompositions are shown in Figure 6-31. Since there are a total of 4 children of `loc1`, we have a total of 4! = 24 possible decompositions. Similar arguments hold for children of `loc3`. Thus,

for the configuration of Figure 6-30 there are a total of 24x2 = 48 decompositions that the matching algorithm of the BPL Tool explores.

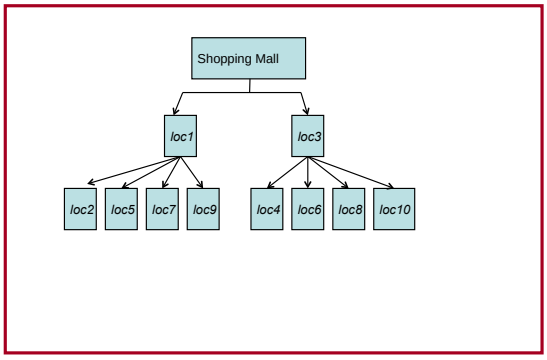

FIGURE 6-30: AN EXAMPLE TOPOLOGY

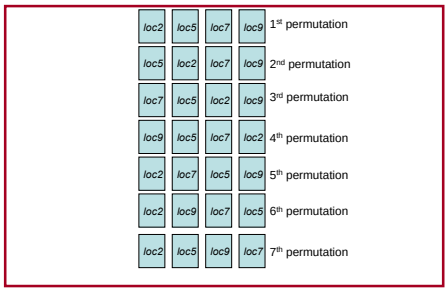

FIGURE 6-31: 7 OF 24 POSSIBLE PERMUTATIONS.

However, in contrast to the prime product, the composition operation between children of a node is interpreted simply as one node being inside another in that order. Consider for example Figure 6-32 where `loc5` is placed inside `loc2` (rather than side-by-side), `loc7` is placed inside `loc5` and `loc9` inside `loc7`. This hierarchy imposes an order on the nodes represented by the sequence <`loc2`,`loc5`,`loc7`,`loc9`> and the Bigraph is decomposed in that order. Thus, there is only one possible decomposition in this case. The matching algorithm of the BPL Tool needs to explore only this single decomposition.

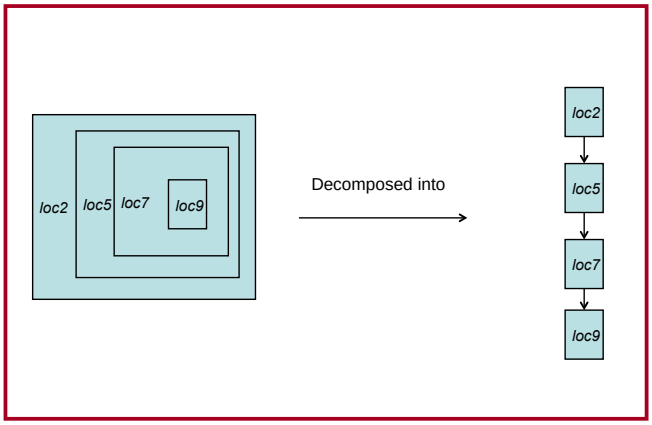

FIGURE 6-32: ONLY ONE POSSIBLE DECOMPOSITION FOR THE COMPOSITION OPERATION.

To show that the response times only grow exponentially for those Bigraphical nodes whose children are all constructed using the prime product, we have devised the following two experiments: We start with the configuration shown in Figure 6-33. Then we successively add children of `loc1` and `loc3` in two different ways. In one experiment, we add children by using the prime product and placing them side-by-side within `loc1` or `loc3` as shown in the left-hand side of Figure 6-33 to Figure 6-39. In the second experiment, we successively add children using the composition operation and placing them one inside another within `loc1` or `loc3` as shown in the right-hand side of Figure 6-33 to Figure 6-39. For each experiment the user only moves between `loc2` and `loc4` and no new device appears in any other location. Thus, between successive experiments, the only thing that changes is the addition of a new location – in the first experiment using prime product operation and in the second experiment using the composition operation.

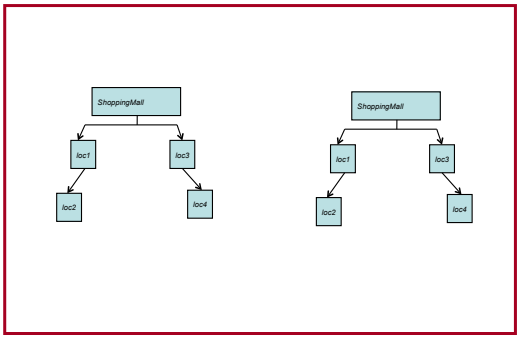

FIGURE 6-33: THE STARTING TOPOLOGY

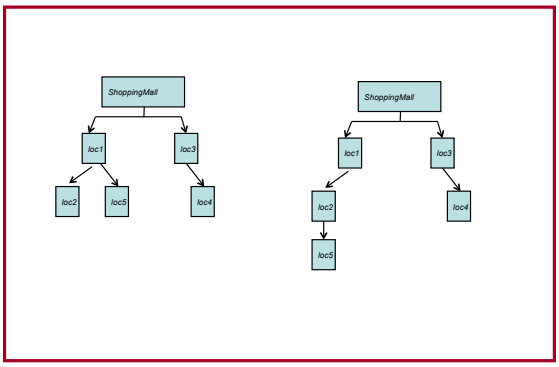

FIGURE 6-34:CHILDREN OF loc1 AND loc3 CONSTRUCTED USING PRIME PRODUCT OPERATION ON THE LEFT AND CONSTRUCTED USING COMPOSITION OPERATION ON THE RIGHT.

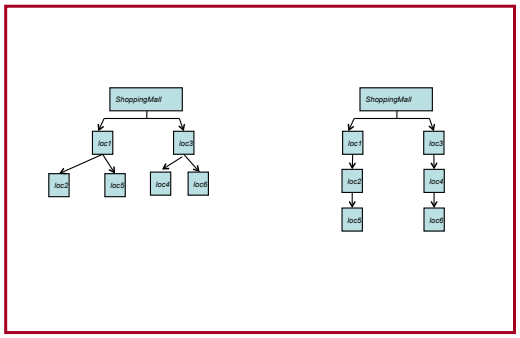

FIGURE 6-35:CHILDREN OF loc1 AND loc3 CONSTRUCTED USING PRIME PRODUCT OPERATION ON THE LEFT AND CONSTRUCTED USING COMPOSITION OPERATION ON THE RIGHT.

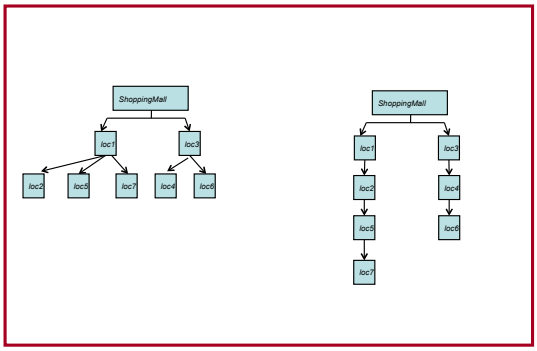

FIGURE 6-36:CHILDREN OF loc1 AND loc3 CONSTRUCTED USING PRIME PRODUCT OPERATION ON THE LEFT AND CONSTRUCTED USING COMPOSITION OPERATION ON THE RIGHT.

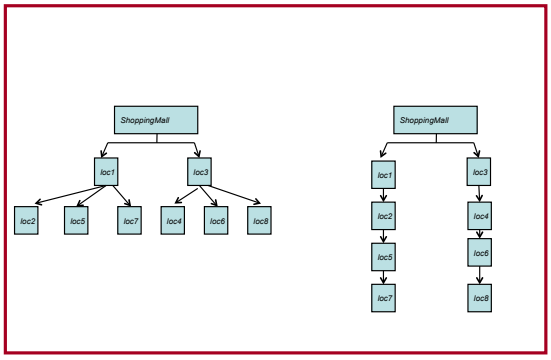

FIGURE 6-37:CHILDREN OF `loc1` AND `loc3` CONSTRUCTED USING PRIME PRODUCT OPERATION ON THE LEFT AND CONSTRUCTED USING COMPOSITION OPERATION ON THE RIGHT.

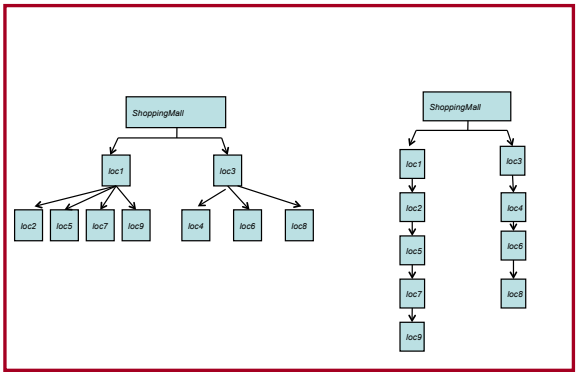

FIGURE 6-38:CHILDREN OF `loc1` AND `loc3` CONSTRUCTED USING PRIME PRODUCT OPERATION ON THE LEFT AND CONSTRUCTED USING COMPOSITION OPERATION ON THE RIGHT.

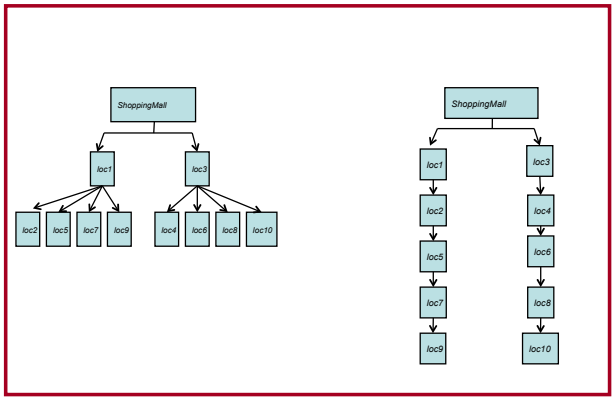

FIGURE 6-39:CHILDREN OF `loc1` AND `loc3` CONSTRUCTED USING PRIME PRODUCT OPERATION ON THE LEFT AND CONSTRUCTED USING COMPOSITION OPERATION ON THE RIGHT.

Calculations for the number of decompositions of `loc1` *and* `loc2` *into their respective children constructed using the prime product operation - all of these decompositions will have to be explored by the matching algorithm*:

1) For the left hand side of Figure 6-33, we have 1! possible decompositions of `loc1` into its children and 1! possible decompositions of `loc3` into its children. Thus, we have a total of 2 decompositions that the matching implementation of the BPL Tool needs to explore.

2) For the left hand side of Figure 6-34, we have 2! possible decompositions of `loc1` into its children and 1! possible decompositions of `loc3` into its children. Thus, we have a total of

$$2+1= 3$$

decompositions that the matching implementation of the BPL Tool needs to explore.

3) For the left hand side of Figure 6-35, we have 2! possible decompositions of `loc1` into its children and 2! possible decompositions of `loc3` into its children. Thus, we have a total of

$$2+2 = 4$$

decompositions that the matching implementation of the BPL Tool needs to explore.

4) For the left hand side of Figure 6-36, we have 3! possible decompositions of `loc1` into its children and 2! possible decompositions of `loc3` into its children. Thus, we have a total of

$$6+2 = 8$$

decompositions that the matching implementation of the BPL Tool needs to explore.

5) For the left hand side of Figure 6-37, we have 3! possible decompositions of `loc1` into its children and 3! possible decompositions of `loc3` into its children. Thus, we have a total of

$$6+6 = 12$$

decompositions that the matching implementation of the BPL Tool needs to explore.

6) For the left hand side of Figure 6-38, we have 4! possible decompositions of `loc1` into its children and 3! possible decompositions of `loc3` into its children. Thus, we have a total of

$$24+6 = 30$$

decompositions that the matching implementation of the BPL Tool needs to explore.

7) For the left hand side of Figure 6-39, we have 4! possible decompositions of `loc1` into its children and 4! possible decompositions of `loc3` into its children. Thus, we have a total of

$$24+24 = 48$$

decompositions that the matching implementation of the BPL Tool needs to explore.

Thus, in general *for this topology*, if there are *n* children representing shops in `loc1` and *m* children representing shops in `loc3` then the number of possible decompositions which the matching implementation of BPL Tool needs to explore is of the order of

$$O(n! + m!)$$

Clearly, the *lower* bound function for the increase in the number of decompositions that the matching implementation of BPL Tool needs to explore as the number of children is increased is exponential.

Calculations for the number of decompositions of `loc1` *and* `loc2` *into their respective children constructed using the composition operation- all of these decompositions will have to be explored by the matching algorithm*:

For the right hand side of all the figures from Figure 6-33 to Figure 6-39, we have 1 possible decompositions of `loc1` into its children and 1 possible decompositions of `loc3` into its children. Thus, we have a total of 2 decompositions that the matching implementation of the BPL Tool needs to explore.

Figure 6-40 and Figure 6-41 show respectively the response times for ‘n’ events and ‘s’ events along with the error bars showing 95% confidence intervals. In these two figures, experiment number 1 corresponds to the situation shown in Figure 6-34 and experiment number 6

corresponds to the situation shown in Figure 6-39. Also, in both the figures, the red bars represent our 'usual' topology where we increase the number of children of `loc1` and `loc3` one-by-one using the prime product operation. This corresponds to the topology shown on the left hand side from Figure 6-34 to Figure 6-39. Similarly, in both the figures Figure 6-40 and Figure 6-41, the yellow bars represent our 'depth first' topology where we increase the number of children of `loc1` and `loc3` one-by-one using the composition operation. This corresponds to the topology shown on the right hand side from Figure 6-34 to Figure 6-39.

These figures show that keeping everything else the same, when we increase the number of nodes, the response times for both 'n' and 's' events increase exponentially for our 'usual' topology. On the other hand, the response times for both 'n' and 's' events remain steady for the 'depth first' topology.

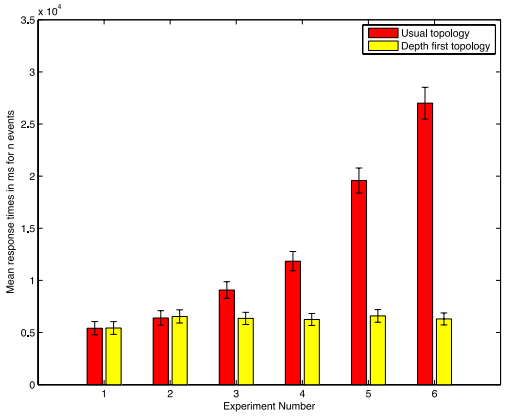

FIGURE 6-40: RESPONSE TIMES OF 'n' EVENT FUNCTION FOR THE EXPERIMENTS WITH THE USUAL TOPOLOGY AND THE EXPERIMENTS CONDUCTED WITH THE DEPTH FIRST TOPOLOGY.

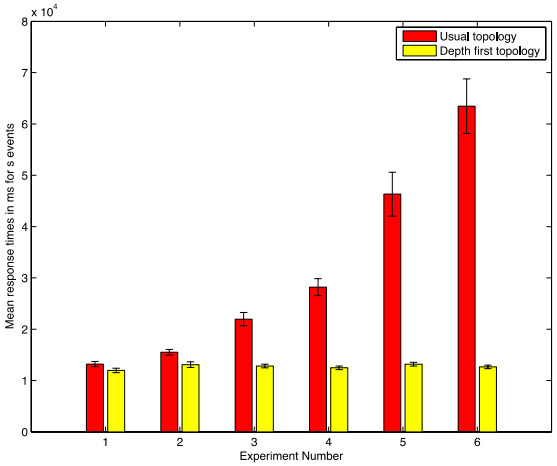

FIGURE 6-41: RESPONSE TIMES OF 's' EVENT FUNCTION FOR THE EXPERIMENTS WITH THE USUAL TOPOLOGY AND THE EXPERIMENTS CONDUCTED WITH THE DEPTH FIRST TOPOLOGY.

On the basis of these results, we conclude that the cause of the exponential increase in the response times in our experiments of section 6.3.3, is the way the BPL Tool's matching algorithm's implementation decomposes the children of a node that have been constructed using the prime product operation. This decomposition produces all the possible permutations of the children of a node. As a result, the matching implementation of the BPL Tool has to explore all these decompositions. Because the number of these decompositions increases at least exponentially, the response times of 'n' and 's' events that use the matching implementation of the BPL Tool also increases at least exponentially.

6.3.5 MEASURING THE EFFECT OF THE WORKLOAD EVENTS ON THE AVAILABLE TIME

We now discuss experimental results that show that even for a favorable scenario, our functions' (`newChangeAmbientScript` and `SCAFaultScript`) response times are more than the available times. Moreover, our results show that the average response times increase exponentially as we increase the number of nodes whereas the average available times remain approximately the same. As discussed earlier, available time is defined as the time interval between the occurrences of consecutive events.

A favorable scenario is depicted in the Figure 6-42. This scenario is favorable because for a workload corresponding to successive string of calls 'nsnsns..' where the events alternate between 'n' and 's', the functions corresponding to these events get as much time as possible to respond. As seen from the Figure 6-42, the available time for a call to 's' is the sum of the time the user pauses between [0s-2s] and half of the walking time till a call to 'n' is made when we reach the right hand side boundary of an ambient in the figure. The available time for 'n' is half of the walking time till a call to 's' is made when we reach the midpoint of an ambient in the figure.

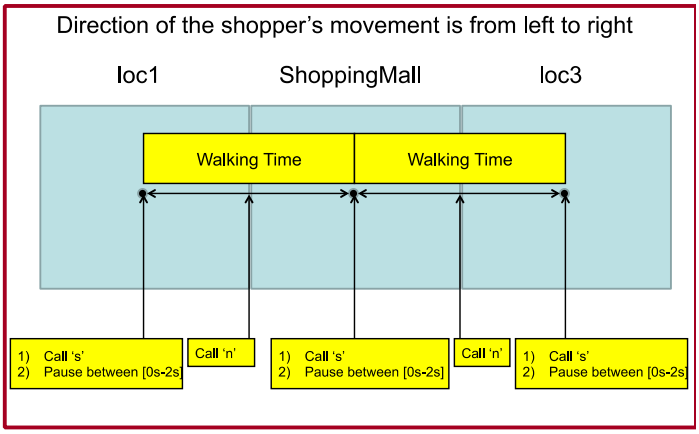

FIGURE 6-42: A FAVORABLE SCENARIO-THE FUNCTIONS GET THE MAXIMUM POSSIBLE TIME TO RESPOND.

We now present an analysis of our experimental data for this favorable scenario using the data generated for each of the three experiments.

(a) *Experiment where we successively increase the number of location nodes by one and run the simulation based on the Shopping mall Mobility model for each of the resulting topology of the shopping mall*:

Figures 6-43 and 6-44 are representative figures that show the data corresponding to experiment number three with the shopping mall having locations up to `loc6`.

In Figure 6-43, we show the response times and available times side-by-side for a given event.

Next, in Figure 6-44, each bar in the graph represents the calculated difference between the response time and available time for experiment number three as the user moves around the shopping mall as per the Shopping Mall Mobility Model. The red bars show the events where response times are more than the available time. On the other hand, the blue bars show that when the user pauses in a shop, the response times of our model at runtime are lesser than the available times and that there are seven such pauses.

Thus, the blue bars show that only when the user pauses in a shop do we get large available times and the response times of our functions are lesser than the available time. This then means, that the BPL Tool's matching algorithm's inefficiency as represented by the red bars needs to be fixed to make our system viable for realistic scenarios.

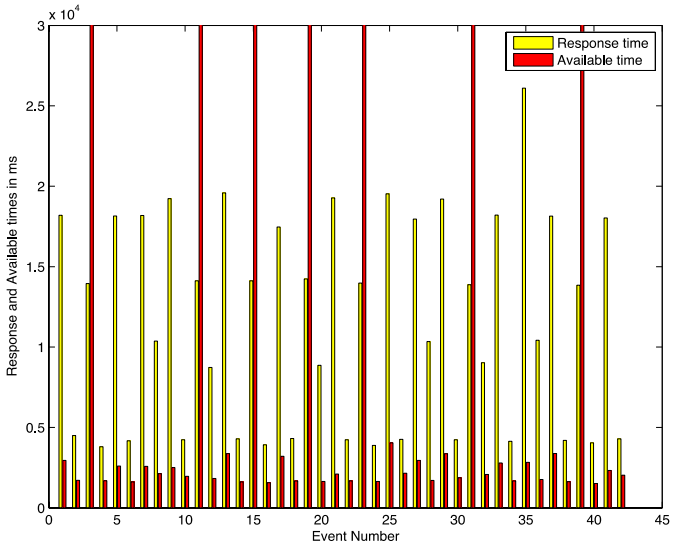

FIGURE 6-43:THE RESPONSE TIMES AND AVAILABLE TIMES FOR EACH EVENT IN MILLISECONDS FOR EXPERIMENT NUMBER 3.

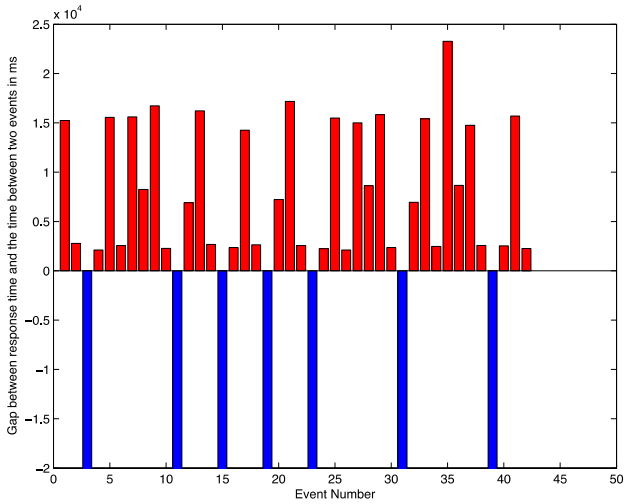

FIGURE 6-44: THE DIFFERENCE BETWEEN THE RESPONSE TIMES AND THE AVAILABLE TIME BETWEEN TWO SUCCESSIVE EVENTS IN MILLISECONDS FOR EXPERIMENT NUMBER 3.

We now summarize the data of this set of experiments for calls to the `newChangeAmbientScript` function (abbreviated as 'n' in the figure) with two figures: Figure 6-45 and Figure 6-46.

The Figure 6-45 shows the mean response times and the mean available times with a 95% confidence interval for each of the two types of means. This figure shows that whereas the mean available time remains approximately constant as we increase the number of nodes, the mean response times increase steadily. This is as it should be since as discussed earlier, the mean available times are generated with the same seeds and same distributions across experiments

whereas the mean response time increases as the implementation of the matching algorithm is inefficient (ITU, 2011), (Birkedal et al., 2007).

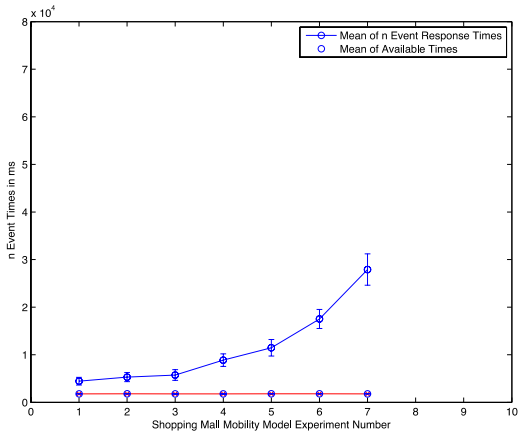

FIGURE 6-45:MEAN OF THE RESPONSE TIMES AND AVAILABLE TIMES FOR 'n' EVENTS SHOWN SEPERATELY FOR EACH OF THE SEVEN EXPERIMENTS.

The Figure 6-46 shows seven mean time gaps with a 95% confidence interval for calls to the `newChangeAmbientScript` function (abbreviated as 'n' in the figure) for each of the seven experiments discussed above. The mean time gap is calculated by taking the mean of the differences between the response times and the available time between two successive events. For calculating the mean we do not include the event when the user pauses in a particular shop. This is because our system gets enough time to respond when the user is pausing in a shop. The response times of our system are problematic (too high) *only* when the user is moving between two target shops (nodes) through intermediate nodes.

This figure also shows an exponential best fitting curve for the seven mean time gaps. This curve has been generated through a regression analysis done using a MATLAB script. The exponential function is given by the expression

$y = be^{mx}$

The values of the constants b and m calculated by the MATLAB script are:

$b = \quad 1.5608e+03$

$m = \quad 0.3836$

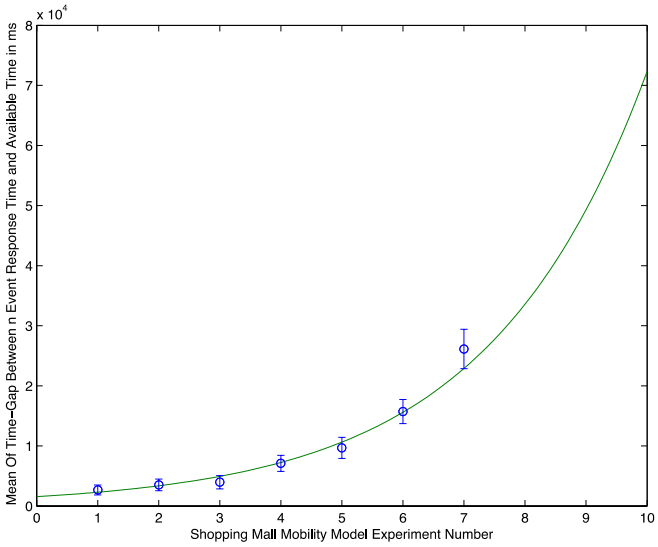

FIGURE 6-46: REGRESSION CURVE OF THE SEVEN MEAN TIME GAPS FOR SEVEN EXPERIMENTS WITH A 95% CONFIDENCE INTERVAL FOR CALLS TO FUNCTION 'n'.

Next, we summarize the data of this set of experiments for calls to the `SCAFaultScript` function (abbreviated as 's' in the figure) for each of the seven experiments discussed above. Similar to the 'n' events above, Figure 6-47 shows that whereas the mean available time remains approximately constant as we increase the number of nodes, the mean response times increase steadily for calls to 's'.

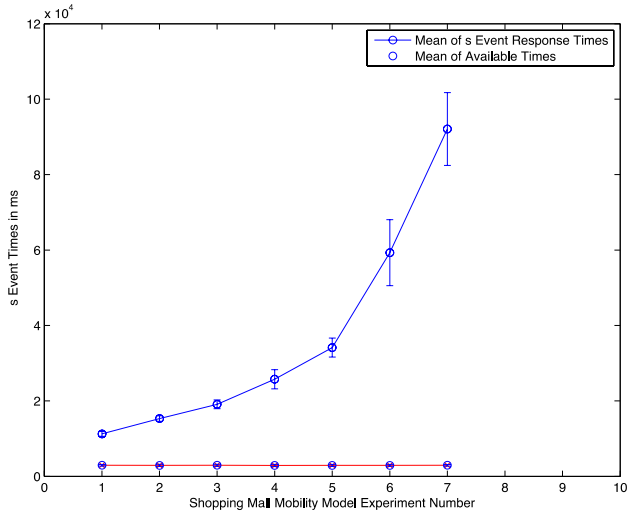

FIGURE 6-47: MEAN OF THE RESPONSE TIMES AND AVAILABLE TIMES FOR 's' EVENTS SHOWN SEPERATELY FOR EACH OF THE SEVEN EXPERIMENTS.

Figure 6-48 shows the seven mean time gaps with a 95% confidence interval for calls to the `SCAFaultScript` function (abbreviated as 's' in the figure) for each of the seven experiments discussed above. Once again, for calculating the mean we do not include the event when the user pauses in a particular shop. This figure also shows an exponential best fitting curve for the

seven mean time gaps. This curve too has been generated through a regression analysis done using a MATLAB script. The exponential function is given by the expression

$y = be^{mx}$

The values of the constants *b* and *m* calculated by the MATLAB script are:

b = 5.3271e+03, m = 0.3858

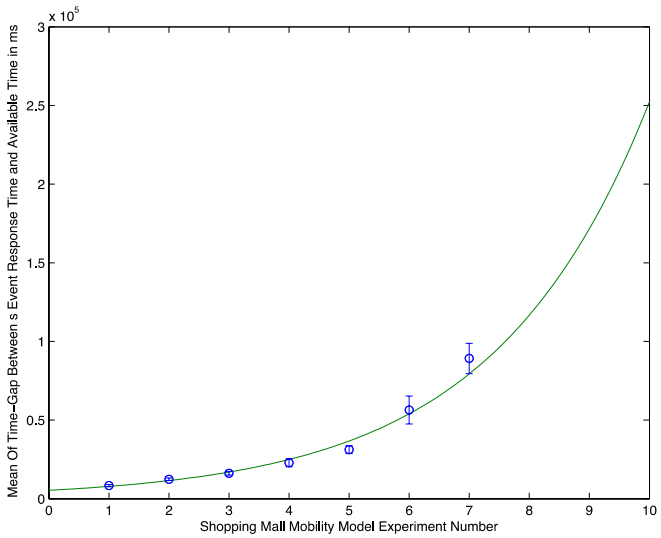

FIGURE 6-48: REGRESSION CURVE OF THE SEVEN MEAN TIME GAPS FOR SEVEN EXPERIMENTS WITH A 95% CONFIDENCE INTERVAL FOR CALLS TO FUNCTION 's'.

(b) *Experiment where one of the three services participating in the service composition never disappears and is available across the shopping mall :* As with the first experiment, Figures 6-49 and 6-50 are representative figures that show the data corresponding to experiment number three with the shopping mall having locations up to `loc6`.

In Figure 6-49, we show the response times and available times side-by-side for a given event.

Next, in Figure 6-50, each bar in the graph represents the calculated difference between the response time and available time for experiment number three as the user moves around the shopping mall as per the Shopping Mall Mobility Model. The red bars show the events where response times are more than the available time. On the other hand, the blue bars show that when the user pauses in a shop, the response times of our model at runtime are lesser than the available times and that there are seven such pauses.

Thus, the blue bars show that only when the user pauses in a shop do we get large available times and the response times of our functions are lesser than the available time. This then

means, that the BPL Tool's matching algorithm's inefficiency as represented by the red bars needs to be fixed to make our system viable for realistic scenarios.

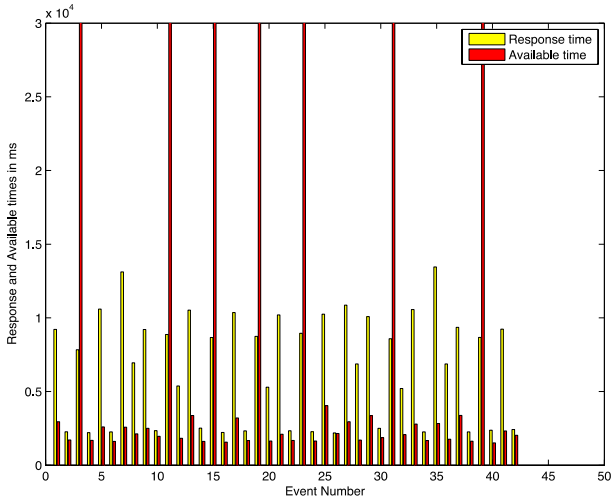

FIGURE 6-49: THE RESPONSE TIMES AND AVAILABLE TIMES FOR EACH EVENT IN MILLISECONDS FOR EXPERIMENT NUMBER 3.

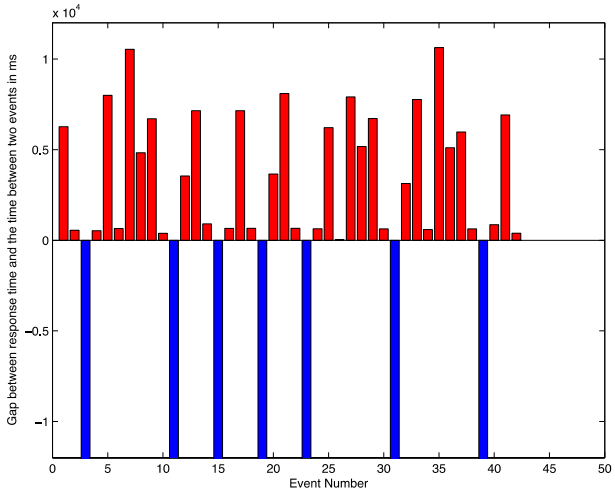

FIGURE 6-50: THE DIFFERENCE BETWEEN THE RESPONSE TIMES AND THE AVAILABLE TIME BETWEEN TWO SUCCESSIVE EVENTS IN MILLISECONDS FOR EXPERIMENT NUMBER 3.

We now summarize the data of this set of experiments for calls to the `newChangeAmbientScript` function (abbreviated as 'n' in the figure) with two figures: Figure 6-51 and Figure 6-52.

The Figure 6-51 shows the mean response times and the mean available times with a 95% confidence interval for each of the two types of means. This figure shows that whereas the mean available time remains approximately constant as we increase the number of nodes, the mean response times increase steadily. This is as it should be since as discussed earlier, the mean

available times are generated with the same seeds and same distributions across experiments whereas the mean response time increases as the implementation of the matching algorithm is inefficient (ITU, 2011),(Birkedal et al., 2007).

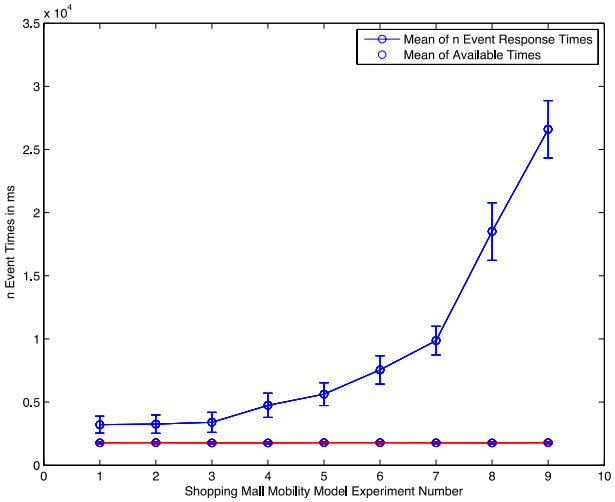

FIGURE 6-51: MEAN OF THE RESPONSE TIMES AND AVAILABLE TIMES FOR 'n' EVENTS SHOWN SEPERATELY FOR EACH OF THE NINE EXPERIMENTS.

The Figure 6-52 shows nine mean time gaps with a 95% confidence interval for calls to the `newChangeAmbientScript` function (abbreviated as 'n' in the figure) for each of the nine experiments discussed above. The mean time gap is calculated by taking the mean of the differences between the response times and the available time between two successive events. For calculating the mean we do not include the event when the user pauses in a particular shop. This is because our system gets enough time to respond when the user is pausing in a shop. The response times of our system are problematic (too high) *only* when the user is moving between two target shops (nodes) through intermediate nodes.

This figure also shows an exponential best fitting curve for the nine mean time gaps. This curve has been generated through a regression analysis done using a MATLAB script. The exponential function is given by the expression

$y = be^{mx}$

The values of the constants b and m calculated by the MATLAB script are:

$b = 1.8578e+03,$

$m = 0.2674$

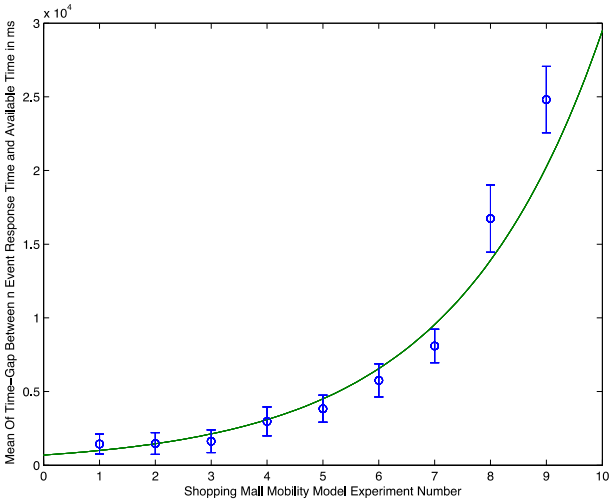

FIGURE 6-52: REGRESSION CURVE OF THE NINE MEAN TIME GAPS FOR NINE EXPERIMENTS WITH A 95% CONFIDENCE INTERVAL FOR CALLS TO FUNCTION 'n'.

Next, we summarize the data of this set of experiments for calls to the `SCAFaultScript` function (abbreviated as 's' in the figures 6-53 and 6-54) for each of the nine experiments discussed above. Similar to the 'n' events above, Figure 6-53 shows that whereas the mean available time remains approximately constant as we increase the number of nodes, the mean response times increase steadily for calls to 's'.

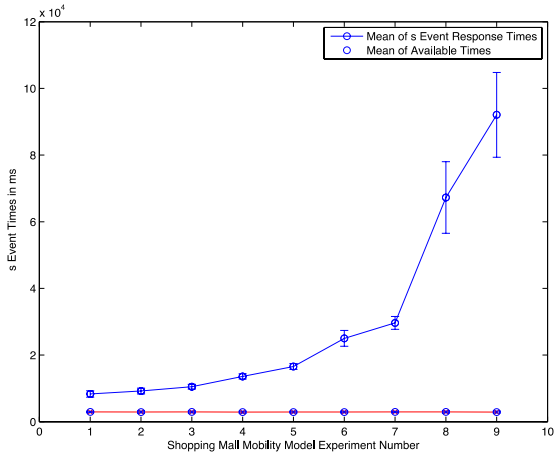

FIGURE 6-53: MEAN OF THE RESPONSE TIMES AND AVAILABLE TIMES FOR 's' EVENTS SHOWN SEPERATELY FOR EACH OF THE NINE EXPERIMENTS.

Figure 6-54 shows the nine mean time gaps with a 95% confidence interval for calls to the `SCAFaultScript` function (abbreviated as 's' in the figure) for each of the nine experiments discussed above. Once again, for calculating the mean we do not include the event when the user pauses in a particular shop. This figure also shows an exponential best fitting curve for the nine mean time gaps. This curve too has been generated through a regression analysis done using a MATLAB script. The exponential function is given by the expression

$y = be^{mx}$

The values of the constants *b* and *m* calculated by the MATLAB script are:

$b = 4.4814e+03, m = 0.2899$

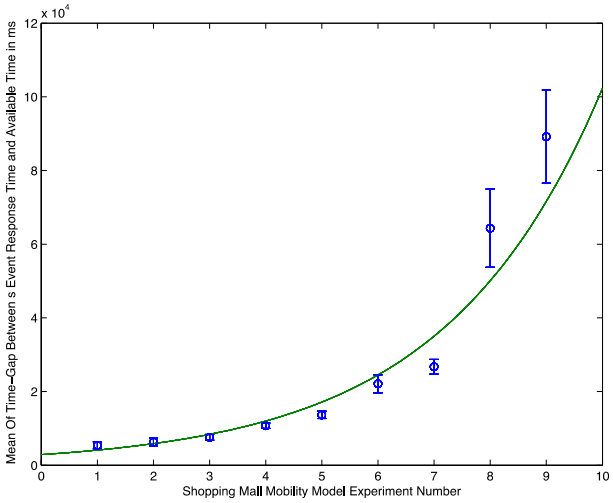

FIGURE 6-54: REGRESSION CURVE OF THE NINE MEAN TIME GAPS FOR NINE EXPERIMENTS WITH A 95% CONFIDENCE INTERVAL FOR CALLS TO FUNCTION 's'.

(c) *Experiment where we successively increase the number of services participating in the composition keeping the number of locations constant*: The first two figures Figure 6-55 and 6-56 are representative figures that shows the data corresponding to experiment number three with five services participating in the composition.

In Figure 6-55, we show the response times and available times side-by-side for a given event.

Next, in Figure 6-56, each bar in the graph represents the calculated difference between the response time and available time for experiment number three as the user moves around the shopping mall as per the Shopping Mall Mobility Model. The red bars show the events where response times are more than the available time. On the other hand, the blue bars show that

when the user pauses in a shop, the response times of our model at runtime are lesser than the available times and that there are six such pauses.

Thus, the blue bars show that only when the user pauses in a shop do we get large available times and the response times of our functions are lesser than the available time. This then means, that the BPL Tool's matching algorithm's inefficiency as represented by the red bars needs to be fixed to make our system viable for realistic scenarios.

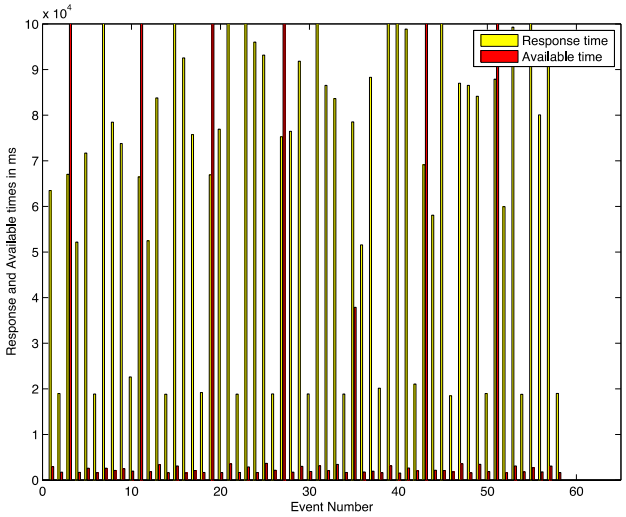

FIGURE 6-55: THE RESPONSE TIMES AND AVAILABLE TIMES FOR EACH EVENT IN MILLISECONDS FOR EXPERIMENT NUMBER 3.

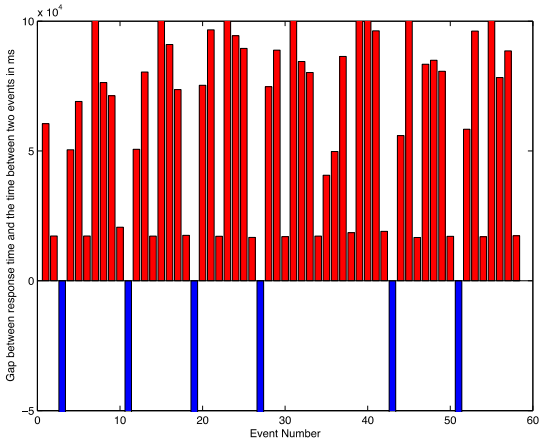

FIGURE 6-56: THE DIFFERENCE BETWEEN THE RESPONSE TIMES AND THE AVAILABLE TIME BETWEEN TWO SUCCESSIVE EVENTS IN MILLISECONDS FOR EXPERIMENT NUMBER 3.

We now summarize the data of this set of experiments for calls to the `newChangeAmbientScript` function (abbreviated as 'n' in the figure) with two figures: Figure 6-57 and Figure 6-58.

The Figure 6-57 shows the mean response times and the mean available times with a 95% confidence interval for each of the two types of means. This figure shows that whereas the mean available time remains approximately constant as we increase the number of nodes, the mean response times increase steadily. This is as it should be since as discussed earlier, the mean available times are generated with the same seeds and same distributions across experiments whereas the mean response time increases as the implementation of the matching algorithm is inefficient (ITU, 2011),(Birkedal et al., 2007).

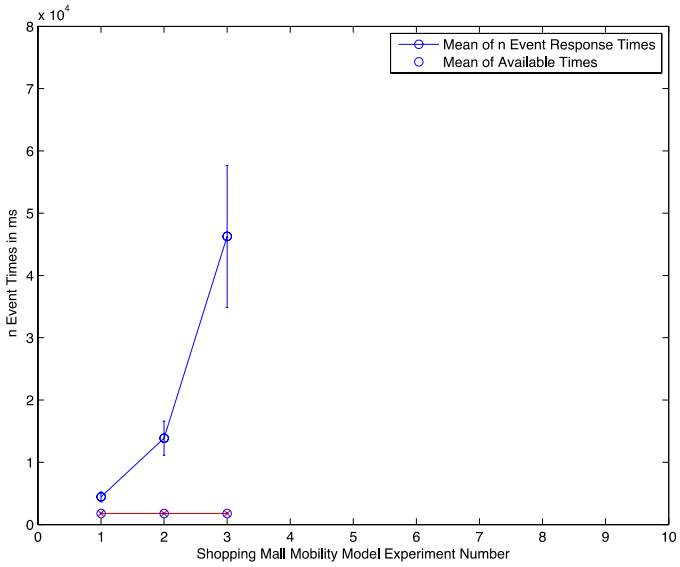

FIGURE 6-57: MEAN OF THE RESPONSE TIMES AND AVAILABLE TIMES FOR 'n' EVENTS SHOWN SEPERATELY FOR EACH OF THE THREE EXPERIMENTS.

The Figure 6-58 shows three mean time gaps with a 95% confidence interval for calls to the `newChangeAmbientScript` function (abbreviated as 'n' in the figure) for each of the three experiments discussed above. The mean time gap is calculated by taking the mean of the differences between the response times and the available time between two successive events. For calculating the mean we do not include the event when the user pauses in a particular shop. This is because our system gets enough time to respond when the user is pausing in a shop. The response times of our system are problematic (too high) *only* when the user is moving between two target shops (nodes) through intermediate nodes.

This figure also shows an exponential best fitting curve for the three mean time gaps. This curve has been generated through a regression analysis done using a MATLAB script. The exponential function is given by the expression

$y = be^{mx}$

The values of the constants b and m calculated by the MATLAB script are:

$b = 674.7071,$

$m = 1.4079$

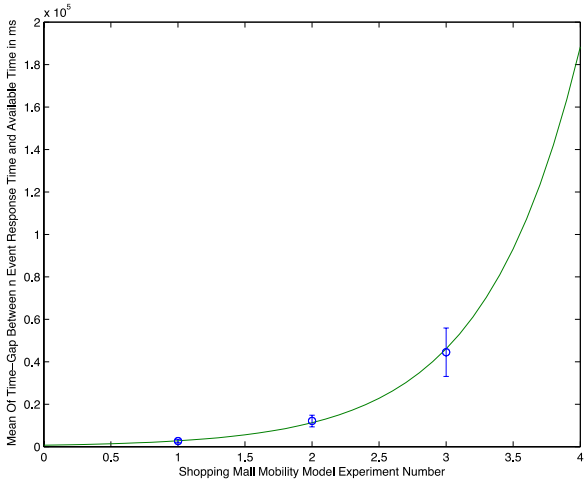

FIGURE 6-58: REGRESSION CURVE OF THE THREE MEAN TIME GAPS FOR THREE EXPERIMENTS WITH A 95% CONFIDENCE INTERVAL FOR CALLS TO FUNCTION 'n'.

Next, we summarize the data of this set of experiments for calls to the `SCAFaultScript` function (abbreviated as 's' in the Figures 6-59 and 6-60) for each of the three experiments discussed above. Similar to the 'n' events above, Figure 6-59 shows that whereas the mean available time remains approximately constant as we increase the number of nodes, the mean response times increase steadily for calls to 's'.

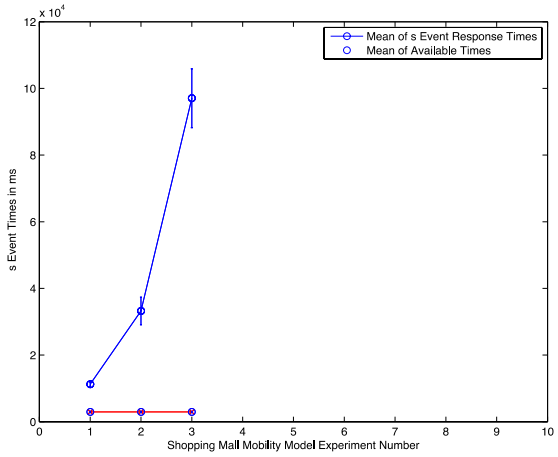

FIGURE 6-59: MEAN OF THE RESPONSE TIMES AND AVAILABLE TIMES FOR 's' EVENTS SHOWN SEPERATELY FOR EACH OF THE THREE EXPERIMENTS.

Figure 6-60 shows the three mean time gaps with a 95% confidence interval for calls to the `SCAFaultScript` function (abbreviated as 's' in the figure) for each of the three experiments discussed above. Once again, for calculating the mean we do not include the event when the user pauses in a particular shop. This figure also shows an exponential best fitting curve for the three mean time gaps. This curve too has been generated through a regression analysis done using a MATLAB script. The exponential function is given by the expression

$y = be^{mx}$

The values of the constants *b* and *m* calculated by the MATLAB script are:

b =2.5424e+03, m = 1.2125

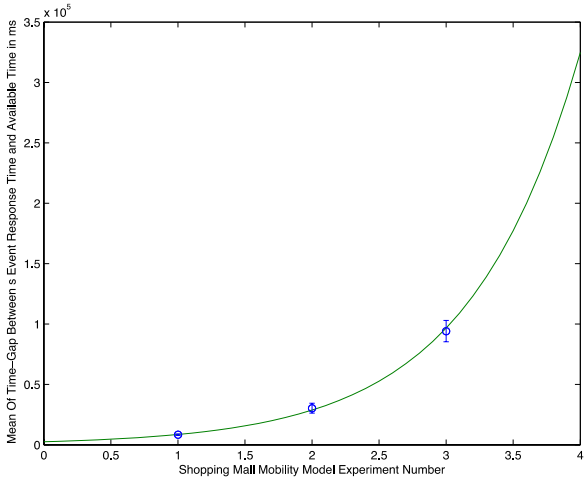

FIGURE 6-60: REGRESSION CURVE OF THE THREE MEAN TIME GAPS FOR THREE EXPERIMENTS WITH A 95% CONFIDENCE INTERVAL FOR CALLS TO FUNCTION 's'.

An unfavorable scenario is depicted in the Figure 6-61. In this case, both the functions 'n' and 's' are called one after another and *there is zero available time to respond*. This is therefore a worst-case scenario. Note too that 'n' and 's' can be called consecutively anywhere in the mall and not necessarily at the mid-point of an ambient as shown in the Figure 6-61. The Figure therefore shows *one* of the many worst-case scenarios.

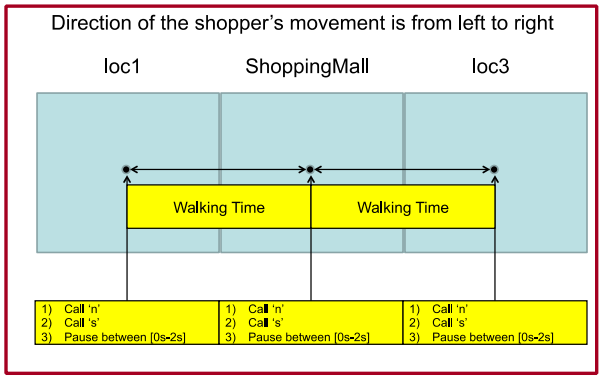

FIGURE 6-61: ONE OF THE MANY WORST CASE SCENARIOS

6.3.6 SUMMARY AND DISCUSSION OF THE EXPERIMENTAL RESULTS

The goal of the three set of experiments discussed in section 6.3.3 was to quantify the effect of the inefficiency of the matching algorithm of the BPL tool on our system. Recall that the matching algorithm has been designed for correctness not efficiency (Elsborg, 2009). The functions offered by this Tool were used by our system to implement the Bigraphical model at runtime. The runtime model's response times have been measured in realistic tests by using the Shopping Mall mobility model as the basis of our simulation.

For each individual experiment in the three sets of experiments discussed in section 6.3.3, we calculated the mean response times with a 95% confidence interval for the function `newChangeAmbientScript`. Thus, for a given set of experiments, we got a set of mean response times each with a 95% confidence interval. We then ran a regression analysis on this set of mean response times to generate a best fitting curve that characterizes their increase as the number of nodes in the Bigraph model increase.

All the three sets of experiments show that as the size of Bigraph is increased, the mean response times increase exponentially.

We then repeated this process for the same set of experiments but this time for the function `SCAFaultScript` and got another best fitting curve from the regression analysis. This curve characterizes the mean response time's increase for the function `SCAFaultScript` as the size of the Bigraph increases for this particular set of experiments.

Once again, all the three sets of experiments show that as the size of Bigraph is increased, the mean response time increases exponentially.

The goal of the two experiments described in section 6.3.4 was to show that the response times of both our functions `newChangeAmbientScript` and `SCAFaultScript` increases exponentially because of the way the BPL Tool's matching algorithm decomposes the children of a node that have been constructed using the prime product operation. In the first experiment of section 6.3.4, we successively added a child location to a location using only the prime product operation. We called this the experiment with the 'usual' topology. In the second experiment of section 6.3.4, we successively added a child location to a location using only the composition operation. We called this the experiment with 'Depth First' topology. We demonstrated that when we use the prime product to add children to a node, the number of possible decompositions by the BPL Tool's matching algorithm increases with an exponential lower bound and the response times of the two functions `newChangeAmbientScript` and `SCAFaultScript` also increase with an exponential lower bound.

Thus, the experiments of section 6.3.3-all of which use the prime product to construct the children representing shops- have established that the naïve implementation of the BPL Tool's matching algorithm is *sufficient* to cause the response times of our functions to increase exponentially despite varying other parameters. On the other hand, the experiments of section 6.3.4 establish that if we *remove* the usage of the prime product to construct the children of a node, and use the composition product instead, the response times of our functions do not increase with an exponential lower bound. As a result we can say that the naïve implementation of the matching algorithm for prime product construction is *necessary* to cause an exponential increase in the response times of our two functions `newChangeAmbientScript` and `SCAFaultScript`. In other words, we have established experimentally that the naïve implementation of the matching algorithm is the *necessary and sufficient cause* for the response times of our two functions to increase exponentially.

In section 6.3.5, we examined one of the many favorable scenarios where our two functions `newChangeAmbientScript` and `SCAFaultScript` get as much time as possible while being called alternately. The results show that the average response times of our functions increase exponentially whereas the average available times remain steady across experiments. As a result, the mean time gap between the response time and available time increases exponentially. Thus, the BPL Tool's matching algorithm's inefficiency needs to be fixed to make our system viable for realistic scenarios.

There are however some limitations of the experiments that we have conducted. Firstly, we did not use any model that was based on real-life data for the distribution of the timings of appearance/disappearance of the services. Only the mobility model of how a shopper moves around in a shopping mall was realistic. Secondly, the laptop that we used to run the SML based

Bigraphical model at runtime was an old and slow machine. As a result, the system keeled off too easily.

Nevertheless, even with these limitations, the goal of the experiments was not to *prove* that as the size of Bigraph is increased, the mean response time increases exponentially for both our functions. The developers of the matching algorithm of the BPL Tool have already acknowledged that their development of the tool was focused on correctness rather than efficiency (Elsborg, 2009) . Instead, what we wanted to show through the experiments was the *effect* of the inefficiency of the matching algorithm of the BPL Tool on our proposal for a practical Bigraphical model at runtime in terms of the shape of a best fitting curve rather than absolute values on a particular machine.

Besides the factors that effect the response times discussed above, there are other *external* factors that might affect the performance of a service composition running on a mobile device. Some of these factors include: delays in discovery of services, the radio communication used, the pattern of mobility, the radio communication used, and the physical characteristics of the settings. These factors were not considered for the experiments described above because in this thesis we wanted to focus only on delays resulting from our implementation of the Bigraphical model at runtime using the BPL Tool.

To the best of our knowledge, our work is the first to evaluate the performance of a system built on top of the BPL Tool using realistic tests. As discussed above, the BPL Tool cannot be used in the current form for a practical implementation of a model at runtime chiefly because the model will not be in-sync with the world in realistic scenarios.

6.4 CONCLUSIONS

In this chapter, we first analyzed our implementation of a Bigraphical model at runtime qualitatively by placing it in the context of the modeling dimensions of self-adaptive software systems. Next, from this qualitative analysis, one of the issues that emerged was the need for a performance evaluation of our Bigraphical model at runtime. For this performance evaluation, we wanted to quantify the effect of the inefficiency of the BPL Tool's matching algorithm on the mean response times of the SML functions of our system.

To quantify this effect of the inefficiency of the BPL Tool with realistic workloads, we designed a test rig. This test rig consisted of an Android machine that ran simulations based on the Shopping Mall Mobility model presented in the paper by Galati et al. (Galati et al., 2013) to generate events that were sent over a TCP connection to a laptop that ran our Bigraphical model

at runtime. These events triggered the execution of appropriate functions in our SML code for the Bigraphical model at runtime.

We established experimentally that the naïve implementation of the matching algorithm is the *necessary and sufficient cause* for the response times of our two functions `newChangeAmbientScript` and `SCAFaultScript` to increase exponentially.

We acknowledge that there are some limitations to the realism of the experiments that we conducted. We did not use any model based on real life data for the distribution of the timings of the appearance/disappearance of services. Moreover, because of the hardware limitations of the laptop that we used to run our SML implementation of the Bigraphical model at runtime, our system would stop responding too easily.

Nevertheless, we did not want to prove that the matching algorithm is inefficient- this has already been acknowledged by the developers of the BPL Tool (Elsborg, 2009) . Instead, we wanted to study the effect of this inefficiency on our system in terms of the shape of a best fitting curve rather than absolute values on a particular machine.

Therefore, within the limitations mentioned above, our experiments show that the BPL Tool cannot be used in the current form for a practical implementation of a Bigraphical model at runtime. This is because the Bigraphical model at runtime will not be in-sync with the world in realistic scenarios if a node has children that are placed side-by-side in the topology being modelled.

In the next chapter, we discuss the implications of the findings of this chapter on our research question.

7 CONCLUSIONS, CONTRIBUTIONS AND FUTURE WORK

7.1 INTRODUCTION

In the past six chapters, we have surveyed the state of art relevant to identify our research question (Chapter 2), defined our research question and established its design implications (Chapter 3), explored the use of Bigraphs and model at runtime as a way to tackle the volatility problem of ubiquitous computing systems (Chapter 4), used the BPL tool to implement a two layered Bigraphical model at runtime (Chapter5), and finally evaluated our implementation both qualitatively and quantitatively.

We now conclude this thesis as follows: In section 7.2, we discuss how our work has answered the research problem posed in Chapter 3, section 3.2. Next in section 7.3, we present a summary of our contributions to new knowledge. Then, in section 7.4, we discuss future work. Finally, in section 7.5, we offer our concluding remarks.

7.2 ANSWERING THE RESEARCH QUESTION

In this section, we discuss how the thesis answers the research question posed in Chapter 3, section 3.2. In particular, we explain why this thesis has shown a way to use Bigraphs at runtime. Furthermore, we explain why this thesis has shown that Bigraphs offer the appropriate language abstractions to address the open research questions being explored by the models at runtime community

In Chapter 3, section 3.2, we described our research question thus:

Are the language abstractions provided by Bigraphs sufficient and appropriate to construct a model at runtime to tackle the problem of volatility in a service composition running on a mobile device?

The two caveats on the scope of the above question were that firstly, we would not replicate all programming language abstraction with our Bigraphical model at runtime-we will abstract upon only some selected elements of the service composition. Secondly, we would be accessing the control constructs of SML through MiniML since Bigraphs lack control structures.

We discuss the answer to the research question using the two evaluation criteria discussed in Chapter 3 and keeping the two caveats discussed above in mind:

Our first evaluation criteria was: Have we been able to *construct* a model at run time that is expressed using Bigraphical abstractions?

We have established in this thesis, that indeed, the language abstractions provided by Bigraphs are sufficient and appropriate to *construct* a model at runtime as a proof-of-concept to tackle the problem of volatility in a service composition running on a mobile device. Notice that unlike the usual practice of using process algebras for simulation, we have used Bigraphs to fire reaction rules in *response* to external events (rather than permitting them to run when they want).

The thesis establishes this claim of constructing a model along the following two dimensions:

1) We have shown how to use Bigraphs to construct a model at runtime,

2) We have also shown that Bigraphs offer the appropriate language abstractions to address the open research questions being explored by the models at runtime community.

We discuss each in turn in Sections 7.2.1.1 and 7.2.1.2.

Our second evaluation criteria was: Can such a Bigraphical model at runtime be in-sync with the real world in terms of the time it takes to respond to the events that are being generated in the real world? Or if they are not in-syc, why not?

In the previous Chapter we have established that for the topology of the shopping mall that we modelled, our Bigraphical model at runtime was not in-sync with the world. The reason for this was that the response time of our model grew exponentially as the size of the model was increased. We showed that this was because of the naïve handling of the prime product children of a node by the BPL Tool's matching algorithm.

We discuss this in greater detail in Section 7.2.2.

7.2.1 USING THE FIRST DIMENSION OF OUR EVALUATION CRITERIA TO TEST IF OUR RESEARCH QUESTION HAS BEEN ANSWERED

The first dimension of our evaluation criteria as discussed in Chapter 3 was:

Have we been able to construct a model at run time that is expressed using Bigraphical abstractions? Such a system will then serve as a proof-of-concept that it is indeed possible to undertake such a construction. This of course will be a constructive proof of existence.

We now discuss the various aspects of the Bigraphical model at runtime that we have constructed. This construction serves as a proof-of-concept that such a system can be designed, implemented and run.

7.2.1.1 USING BIGRAPHS TO CONSTRUCT A MODEL AT RUNTIME

1) Instead of using Bigraphs to model systems for simulation, we have used Bigraphs to express a model that is *causally* connected to a running system. This causal connection is established by associating Bigraphical reaction rules with system events corresponding to 'Observed effects' of Chan et al.(Chan et al., 2007b) in Section 4.4.2.5. Moreover, we have also associated input events with adaptation commands to be sent back to the system (Section 5.2.3).

2) We have discussed the techniques to use Bigraphical abstractions mapped to a programming language to model at runtime a real-world system. Our contribution has been to show a way to take adaptation decisions at runtime based on the current configuration of the two layers expressed in Bigraphical abstractions mapped to MiniML (Section 5.4). Moreover, modeling reconfigurations through reaction rules has enabled us to handle run-time complexity by having a few rules intensionally construct the set of infinitely many possible re-configurations of the system (Sections 4.4.1 and 4.4.2).

3) We have shown a way Bigraphical abstractions can be used to implement standard system techniques like caching, delayed-write, pre-fetching. Our insight is that a model at runtime expressed with Bigraphs can be used as a software cache to store pre-fetched location and id of devices, which are offering backup of those services that are participating in the composition (Section 4.4). The Bigraphical model also caches the current state of the service composition. Using a model at run time in such a way forms the core of our strategy to deal with volatility. Furthermore, we show how to use reaction rules to deal with the arbitrary order of runtime events. This is possible because reaction rules can be fired in any arbitrary order in our system (Section 5.4).

7.2.1.2 THE APPROPRIATE BIGRAPH ABSTRACTIONS TO TACKLE MODELS AT RUNTIME'S RESEARCH QUESTIONS

1) We have shown how a runtime model can provide a means to store and retrieve information about the environment and the system. We achieve this by representing two views of the system in two layers of our Bigraphical model. The WORLD layer caches the location of devices (Section 4.4.1). The Service Composition Architecture (SCA) layer models the structure of the composition and the state of each service participating in it (Section 4.4.2).

2) To facilitate adaptation, we have demonstrated how a Bigraphical model at runtime can provide a representation of the current state and reconfiguration rules. The state of our two-layered Bigraphical model represents the current state of the system. Reaction rules at the SCA

layer model the internal state changes in the composition (Section 4.4.2). Reaction rules at the WORLD layer model the changes in the state of the external environment due to volatility (Section 4.4.1).

3) We have described how a runtime model can enable us to reason about operating environment and runtime behaviour to determine an appropriate form of adaptation. This is achieved by querying the WORLD layer of our Bigraphical model at runtime for information about the operating environment (Section 5.3.2). Similarly, we can query the SCA layer of our Bigraphical model about the runtime behaviour by retrieving the state of any service of the composition (Section 5.3.2). This information about the WORLD and SCA layer can then be used by our system to choose what to do next (Section 5.4). Notice however as we show in the last chapter, it is not possible for the information to be up-to-date given the current implementation of the BPL Tool's matching algorithm.

4) We have shown how a runtime model can provide meta-information along the following two dimensions: a) efficient use of time, b) location dependency. For providing meta-information so as to make efficient use of time, we use a pre-fetching strategy to cache locations of services' devices in our Bigraphical runtime model before they stop working (Section 4.3 and Section 4.4). These cached locations of devices constitute the necessary meta-information. For providing meta-information to deal with location dependency of services, we store the location of services in the WORLD layer of our Bigraphical model that can capture the containment topology of the shopping mall (Section 4.4.1). These cached locations of services constitute the necessary meta-information.

5) We have shown a way to combine two models of runtime into one. When an event occurs at the SCA layer through the triggering of a Bigraphical reaction rule of that layer, a corresponding Bigraphical reaction rule is triggered at the WORLD layer (Section 4.4). The two models are required because for the same system, we need to manage two separate concerns- the internal structure of the service composition and the external environment in which the composition is running. Each of the two concerns is modelled separately by the two models of runtime.

6) We have shown a way to select one appropriate adaptation command out of many such commands at runtime. We use Bigraphical reaction rules to transform a malfunctioning service composition back into a working one. Each of the reaction rules that changes any malfunctioning state of a service back into the 'Working' state corresponds to an adaptation command sent back to the running system (Section 5.2, Section 5.4).

We have now shown why this thesis has successfully answered the research question posed in Chapter 3, section 3.2 using the first evaluation criteria.

7.2.2 USING THE SECOND DIMENSION OF OUR EVALUATION CRITERIA TO TEST IF OUR RESEARCH QUESTION HAS BEEN ANSWERED

The second dimension of our evaluation criteria as discussed in Chapter 3 was:

Can such a Bigraphical model at runtime be in-sync with the real world in terms of the time it takes to respond to the events that are being generated in the real world? Or if they are not in-syc, why not? One of the ways we could do this is by building a test-rig, which will load our Bigraphical model at run time with appropriate events and measure its response times.

As discussed in Chapter 6, we designed a test rig where an Android machine running simulations based on Galati et. al.'s Shopping Mall Mobility model (Galati et al., 2013) generated workload events. These events were sent over a TCP connection to our Bigraphical model at runtime executing on a laptop. The Bigaphical model at runtime responded by sending appropriate commands back to the Android machine.

This test rig ran our simulation experiments that have established the following within experimental limits:

i. Response times of our Bigraphical model at runtime increase exponentially as we increase the number of Bigraphical nodes of the model.
ii. The necessary and sufficient cause for such an exponentially increasing running time in our model is the naïve handling of the decomposition of the Bigraphical prime product children of a node by the BPL Tool's matching algorithm.

The experimental limits were as follows:

i. In our simulations, the distribution of the timings of appearance and disappearance of services was not based on any real-life data.
ii. The laptop that we used to run our Bigraphical model at runtime was old and slow and so our system stopped responding too easily.

Notwithstanding these limitations, given the current implementation of the BPL tool, we concluded in Chapter 6 that the Bigraphical model at runtime would not be in-sync with the real world for realistic scenarios if Bigraphical nodes have children that are placed side-by-side.

Some other external factors effecting a service composition running on a mobile device were not considered. This was because the scope of the thesis was limited only to delays resulting from our implementation of the Bigraphical model at runtime using the BPL Tool. Some of these unconsidered factors include: delays in discovery of services, the radio communication

used, the pattern of mobility, the radio communication used, and the physical characteristics of the settings etc.

7.3 CONTRIBUTIONS TO KNOWLEDGE

Our contributions to knowledge stem from answering our research question as discussed in the previous section. In particular the ideas embodied in our implementation of the model at runtime with Bigraphs constitute our contribution. Also, our implementation has led to a qualitatively increased understanding of how to use Bigraphs in a practical application and how to design a model at runtime for a volatile system.

We now discuss the contributions in detail:

1) This thesis responds to a call by Robin Milner (Milner, 2009) to explore the appropriateness of using Bigraphs in practical applications. We have constructed a model at runtime with Bigraphs. In particular our contributions are:

a) We have shown that Bigraphs can express models at runtime that are causally connected to a running system. In other work, Bigraphs have been used for simulation.
b) We have shown that Bigraph's reaction rules can be used to model runtime complexity by having a few rules intensionally construct the set of infinitely many possible re-configurations of the system. Dealing with infinitely many possible re-configurations of the system in this manner helps us in avoiding the inherent state-space explosion problem associated with Morin's (Morin et al., 2008) work where infinite variants of a model are needed to represent infinitely large possible model states.
c) We have shown a way Bigraphical abstractions can be used to implement standard system techniques like caching and pre-fetching. Such usage of Bigraphs is a novel contribution to the debate in the research community about Bigraphs' practicability.

2) This thesis also responds to the research questions posed by the models at runtime community at the Dagstuhl seminar (Aßmann et al., 2012) by offering Bigraphical language abstractions as an appropriate solution. In particular, to the best of our knowledge, we have demonstrated for the first time the following:

a) The use of Bigraphs to express a runtime model that provides a means to store and retrieve information about the environment and the system.
b) The use of Bigraphs to express a runtime model to facilitate adaptation by providing a representation of the current state and reconfiguration rules.

c) How a runtime model expressed with Bigraphs can enable us to reason about the operating environment and runtime behaviour to determine an appropriate form of adaptation.
d) The use of a runtime model expressed with Bigraphs that can provide meta-information along the following two dimensions: i) efficient use of time, ii) location dependency.
e) The use of Bigraphs to combine two models of runtime into one.
f) The use of Bigraphical reaction rules to select one appropriate adaptation command out of many such commands at runtime.

3) The general lessons learnt from using Bigraphs for a practical application such as a model at runtime are (because these are generalisations, by definition they are speculative and need further research):

a) We have used the reaction rules of Bigraphs to model transformation of one architectural configuration to another. This is one of the many ways in which rules are used in a rule-based programming language (Wagner et al., 2004). More generally therefore, Bigraphs can perhaps be used as a *basis for designing* rule-based programming language for example in reasoning and machine learning systems. This is unlike the way reaction rules are used traditionally in process algebras as specifications for their semantics and needs further investigation.
b) We have modelled the environment view in the WORLD layer and the system view in the SCA layer of our Bigraphical model at runtime. Therefore, in general, Bigraphs can perhaps be investigated for knowledge representation of the environment and the system. That is to say, a structured representation (Russell and Norvig, 2010) of the state of the environment and system might be possible with Bigraphs where a state includes objects with attributes linked to other objects.
c) We have demonstrated that it is possible to implement a system expressed in Bigraphs without appealing to the underlying mathematical theory by using the BPL Tool's API as an *interface* to Bigraphs. As a programmer, we didn't need to worry about the mathematically correct *implementations* of those BPL interfaces. Therefore, the use of Bigraphs can perhaps gain acceptance in the larger programming community. As Parnas points out, mathematical nature of formal theory is often perceived by software developers as useless (Parnas, 2010):

   "..most software developers perceive formal methods as useless theory that has no connection with what they do. There is no quicker way to lose the attention of a room full of programmers than to show them a mathematical formula. Developers see the need for improvement and will try almost any new method-provided it does not look like mathematics"

4) The general lessons learnt from our experiences of designing models at runtime are (because these are generalisations, by definition they are speculative and need further research):

a) The faults of service composition that our Bigraphical model at runtime needs to respond to can occur in an order that might or might not follow a specific probability distribution. In general, runtime phenomena are ill-structured in that there is no pre-determined order of runtime events (Because reaction rules can be fired in any order, they are an appropriate abstraction).

b) The configuration of a service composition can change infinitely many times at runtime. Therefore, we have used reaction rules to intensionally generate these reconfigurations. Thus, intensional techniques are a possible approach to express infinite reconfigurations at runtime.

c) In our implementation, we have shown a way to meta-program the reaction rules through abstraction by parameterisation so that matching algorithm of the tool returns a single match giving us the ability to dynamically program the model at runtime. The same meta-programming techniques have also been used by us to show a way to generate infinitely many reaction rules at runtime from a single function. Moreover, we show a way to query the Bigraph structure. In general therefore, languages used could possibly support meta-programming to parametrize ,code, components etc. Also, compositional query mechanisms similar to SQL could also be provided.

d) We have used our Bigraphical model at runtime to cache pre-fetched information. In general, abstractions used to express models at runtime should be able to support classic systems techniques to deal with high latency, low throughput and bottlenecks (e.g. caching pre-fetched information).

e) Our implementation assumes that that there is an exception handling mechanism associated with the service composition in a layer underneath our system. Within our system boundary, we have modelled the effects of volatility on the service composition that trigger those exception-handling mechanisms. Hence generally, volatility management in runtime models could be factored out from the traditional exception handling mechanisms and be dealt with at a higher level of abstraction.

f) In our system, each event (modeled by a reaction rule) that comes in is associated with corresponding adaptation command that is produced by our system as output. Thus in general, Bigraphs provide us with the useful abstraction of reaction rules that like our implementation can be used to deal with complex runtime events and to associate commands and policies with those events.

5) We have implemented the model at runtime using the BPL Tool (ITU, 2011). We suggest the following enhancements for BPL Tool:

a) The BPL Tool's matching algorithm has been written for correctness rather than efficiency. This has meant an exponential increase in the response times of our system. This increase is caused by a naïve handling of the decomposition of the prime product children of a node by the matching algorithm of the BPL Tool. We suggest that this problem with the prime product children be resolved.
b) It would be very useful if in-built functions in BGVal module are provided to modify, traverse (e.g. findChild, findParent functions that we had to develop in our implementation) and update Bigraphs that can then be used by programmers not necessarily interested in the mathematical properties of the structure of the Bigraph.
c) We did not use links of Bigraphs as they are not supported by the tool's matching algorithm. The matching algorithm should support Bigraphs with links.
d) We have modelled runtime events in our implementation with Bigraphical reaction rules. We have assumed that these events are inputted to our system serially. However, runtime events can occur concurrently and the tool does not support concurrent firing of the reaction rules. Therefore, concurrent access to Bigraph structures and support for concurrent firing of reaction rules is needed for a runtime system.

Thus, we have now demonstrated that this thesis has made a significant contribution to knowledge.

7.4 FUTURE WORK

Now that we have successfully demonstrated the appropriateness of using Bigraphs to express a model at runtime to tackle the problem of volatility in a service composition running on a mobile device, we discuss the future directions in which this thesis's work can be taken.

1) Our architecture which supports a model at runtime and our choice of language-Bigraphs-which is a meta-process algebra (Birkedal et al., 2006) shows a way in which we could leverage the verification and validation framework for system assurance of self-adaptive systems by Cheng et al. and Lemos et al. (Cheng et al., 2009, Lemos et al., 2012). Consider Figure 7-1 which shows Cheng et. al.'s framework for adaptive system assurance. A system transitions through a series of operational modes represented by index j. A system in an operational mode j can undergo a sequence of adaptations. Each item in the sequence is represented by a context-system state configuration. For example, $(C + S)_{ji}$ denotes the i^{th} combination of s context-system state. This state is part of a sequence where the sequence itself depends of the requirements of system mode j. The transition t_{j0} represents context or system changes. The model that corresponds to the configuration $(C + S)_{ji}$ is shown as M_{ji} in the Figure 7-1. In the same figure, m_{ji} represents the evolution of a model from one configuration to

another. Each of this m_{ji} could possibly be represented by a reaction rule and a configuration of the model could be represented by a Bigraphical structure.

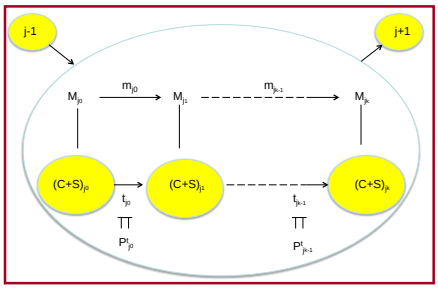

FIGURE 7- 1: VERIFICATION AND VALIDATION MODEL WHERE A SYSTEM IN OPERATIONAL MODE j UNDERGOES A SEQUENCE OF ADAPTATIONS.

2) Our architecture paves the way to link an equivalence checker and a model checker (Perrone et al., 2012) at the goal management layer (Kramer and Magee, 2007) to a runtime model at the change management layer (Kramer and Magee, 2007). For example, we could pre-fetch only those services, which have been checked to be 'equivalent' to the malfunctioning services using equivalence checker. In this, we assume that when the service composition is formed outside our system boundary and that it follows a specification that describes the participating services formally. Also, we could verify before a fault occurs that the resulting service composition will satisfy some desired property using the model checker and store this information as part of our strategy of caching pre-fetched information. An issue that needs to be tackled is that of no equivalent replacement services being available to replace a malfunctioning service. The appropriate mechanisms and strategies to use equivalence checkers and model checkers in such a situation needs to be investigated keeping in mind the inherent un-decidable nature of the problem.

3) We need a way to compare performance of our implementation of a model at runtime with others. To the best of our knowledge, no work has explored the issues that need to be addressed for such a comparison. If our model at runtime is looked upon as black box where service composition faults are input as events and appropriate adaptation commands are the output, then we could define response time of our system as the time interval between inputting of events and outputting of adaptation commands. When our model at runtime is operating at its capacity, a queue of events will build up. An event's waiting time could be estimated using queuing theory (Saltzer and Kaashoek, 2009, Jain, 1991). In queuing theory, the time it takes to process a request is called service time and the rate of arrival of requests for service is known as offered

load. For a runtime model, we hypothesize that the event arrival distribution will be 'bursty' (Barabási, 2010). How the service time varies for different implementations of models at runtime is a question for future work. A related issue is that of guaranteeing the timeliness of the adaptation mechanisms to deal with a high rate of malfunctioning of services in a volatile environment.

4) Similar to other systems like computer networks, we want our model at runtime to be responsive (i.e. low response time) and stable (our model's response should not go into uncontrollable oscillations due to some inputs) (Keshav, 2012). Thus, in the terminology of feedback control systems (Hellerstein et al., 2004), our model at runtime is the target system.

Feedback control theory defines control input to a target system as *"a parameter that affects the behavior of the target system and can be adjusted dynamically"* (Hellerstein et al., 2004). A disturbance input to a target system is defined as *"any change that affects the way in which the control input influences the measured output"* (Hellerstein et al., 2004). Furthermore, a measured output of the target system is defined as *"a measurable characteristic of the target system"* (Hellerstein et al., 2004). Finally a noise input to a target system is defined as *"any effect that changes the measured output produced by the target system"* (Hellerstein et al., 2004).

We could define our control input as the number of cached services that we are maintaining in the model at runtime for each service. We could vary the number of services that we pre-fetch and cache. Also, our disturbance input could be the high rate of events (each event corresponds to a fault in the service composition) our model at runtime should respond to. This is because the control input i.e. the number of cached service would need to be increased for example if the rate of events increased. Furthermore, our measured output could be our model's response time. Finally, the noise input to our model could be for example due to the varying number of other applications besides the service composition that the user could be running on her mobile device.

We could then use feedback control techniques to vary the number of services we are caching in our model for each service depending on the number of events being inputted to our system. How this usage of feedback control techniques results in maintaining our models at runtime's steady response rate is a question worth exploring for future work.

7.5 CONCLUDING REMARKS

In this thesis, we have demonstrated the appropriateness of the language abstractions of Bigraphs to construct a model at runtime to tackle the problem of volatility in a service

composition running on a mobile device. We have shown how to express a model at runtime with Bigraphs. Moreover, Bigraphical abstractions have been used by us to address the open research questions being explored by the models at runtime community. Since we have shown a way to use Bigraphs without appealing to the underlying mathematical concepts, it becomes easier for the usage of Bigraphs to be accepted by a wider audience. Moreover, our work paves the way for a further exploration of Bigraphical abstractions by the researchers working on issues related to models at runtime. Finally, our work is a first step to show that Bigraphs can survive serious experimental application as required by Milner (Milner, 2008b):

The Bigraphical model "*is only a proposal; it can only become foundational model for ubiquitous computing if it survives serious experimental application".*

BIBLIOGRAPHY

The following is the list of references used:

ACETO, L., INGÓLFSDÓTTIR, A., LARSEN, K. G. & SRBA, J. 2007. *Reactive systems: modelling, specification and verification*, Cambridge University Press.

ALONSO, G., CASATI, F., KUNO, H. & MACHIRAJU, V. 2010. *Web Services: Concepts, Architectures and Applications*, Springer Publishing Company, Incorporated.

ALUR, R. & DILL, D. L. 1994. A theory of timed automata. *Theor. Comput. Sci.,* 126**,** 183-235.

ANDERSSON, J., DE LEMOS, R., MALEK, S. & WEYNS, D. 2009. Modeling dimensions of self-adaptive software systems. *Software Engineering for Self-Adaptive Systems***,** 27-47.

APACHE. 2014. *Commons Math: The Apache Commons Mathematics Library* [Online]. Available: http://commons.apache.org/proper/commons-math/ [Accessed July 2014].

ASSMAN, U., BENCOMO, N., CHENG, B. H. C. & FRANCE, R. B. 2012. *Models@run.time (Dagstuhl Seminar 11481)* [Online]. Dagstuhl, Germany: Schloss Dagstuhl--Leibniz-Zentrum fuer Informatik. Available: http://drops.dagstuhl.de/opus/volltexte/2012/3379/ [Accessed 15 November 2012].

AVIZIENIS, A., LAPRIE, J. C., RANDELL, B. & LANDWEHR, C. 2004. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on,* 1**,** 11-33.

BARABÁSI, A. L. 2010. *Bursts: The hidden pattern behind everything we do*, EP Dutton.

BARDRAM, J. & FRIDAY, A. 2010. Ubiquitous Computing Systems. *In:* KRUMM, J. (ed.) *Ubiquitous Computing Fundamentals*

. 1 ed.: CRC Press.

BARNES, D. J. & KOLLING, M. 2013. *Objects first with Java: A practical introduction using Bluej*, Pearson Prentice Hall London.

BARR, M. & WELLS, C. 1990. Category theory for computing science. *Prentice-Hall,* 212**,** 222.

BARROS, A., DUMAS, M. & TER HOFSTEDE, A. 2005. Service interaction patterns. *Business Process Management***,** 302-318.

BENCOMO, N. 2009. On the use of software models during software execution. *Proceedings of the 2009 ICSE Workshop on Modeling in Software Engineering.* IEEE Computer Society.

BENCOMO, N., GRACE, P., FLORES, C., HUGHES, D. & BLAIR, G. 2008. Genie. Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on, 2008. IEEE, 811-814.

BENNACEUR, A., FRANCE, R., TAMBURRELLI, G., VOGEL, T., CAZZOLA, W., COSTA, F. M., PIERANTONIO, A., TICHY, M., AKSIT, M. & EMMANUELSON10, P. 2014. Mechanisms for Leveraging Models at Runtime.

BIRKEDAL, L., DAMGAARD, T. C., GLENSTRUP, A. J. & MILNER, R. 2007. Matching of Bigraphs. *Electron. Notes Theor. Comput. Sci.,* 175**,** 3-19.

BIRKEDAL, L., DEBOIS, S., ELSBORG, E., HILDEBRANDT, T. & NISS, H. 2006. Bigraphical models of context-aware systems. *Foundations of Software Science and Computation Structures, Proceedings,* 3921**,** 187-201.

BLACKWELL, C. 2011. Formally modeling the electricity grid with bigraphs. Proceedings of the Seventh Annual Workshop on Cyber Security and Information Intelligence Research, 2011. ACM, 23.

BLAIR, G., BENCOMO, N. & FRANCE, R. B. 2009. Models@ run.time. *Computer,* 42**,** 22-27.

BRAMBILLA, M., CABOT, J. & WIMMER, M. 2012. Model-driven software engineering in practice. *Synthesis Lectures on Software Engineering,* 1**,** 1-182.

BRONSTED, J., HANSEN, K. M. & INGSTRUP, M. 2010. Service composition issues in pervasive computing. *Pervasive Computing, IEEE,* 9**,** 62-70.

CACERES, R. & FRIDAY, A. 2012. Ubicomp Systems at 20: Progress, Opportunities, and Challenges. *IEEE Pervasive Computing,* 11**,** 14-21.

CALDER, M. & SEVEGNANI, M. 2012. Process algebra for event-driven runtime verification: a case study of wireless network management. Integrated Formal Methods, 2012. Springer, 21-23.

CAPORUSCIO, M., DI MARCO, A. & INVERARDI, P. 2007. Model-based system reconfiguration for dynamic performance management. *Journal of Systems and Software,* 80**,** 455-473.

CAZZOLA, W., GHONEIM, A. & SAAKE, G. 2004. Software Evolution through Dynamic Adaptation of Its OO Design. *Objects, Agents, and Features***,** 31-48.

CHAKRABORTY, D., JOSHI, A., FININ, T. & YESHA, Y. 2005. Service composition for mobile environments. *Mob. Netw. Appl.,* 10**,** 435-451.

CHAN, K. S., BISHOP, J., STEYN, J., BARESI, L. & GUINEA, S. 2007a. A Fault Taxonomy for Web Service Composition. *In:* ELISABETTA, N. & MATEI, R. (eds.) *Service-Oriented Computing - ICSOC 2007 Workshops.* Springer-Verlag.

CHAN, K. S. M., BISHOP, J., STEYN, J., BARESI, L. & GUINEA, S. 2007b. A Fault Taxonomy for Web Service Composition. *In:* NITTO, E. & RIPEANU, M. (eds.) *Service-Oriented Computing - ICSOC 2007 Workshops.* Springer Berlin Heidelberg.

CHANG, Z., MAO, X. & QI, Z. 2007. An Approach based on Bigraphical Reactive Systems to Check Architectural Instance Conforming to its Style. Theoretical Aspects of Software Engineering, 2007. TASE'07. First Joint IEEE/IFIP Symposium on, 2007. IEEE, 57-66.

CHANG, Z., MAO, X. & QI, Z. 2008a. Formal analysis of architectural policies of self-adaptive software by bigraph. Young Computer Scientists, 2008. ICYCS 2008. The 9th International Conference for, 2008a. IEEE, 118-123.

CHANG, Z., MAO, X. & QI, Z. 2008b. Towards a Formal Model for Reconfigurable Software Architectures by Bigraphs. *Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008).* IEEE Computer Society.

CHENG, B. H., EDER, K. I., GOGOLLA, M., GRUNSKE, L., LITOIU, M., MÜLLER, H. A., PELLICCIONE, P., PERINI, A., QURESHI, N. A. & RUMPE, B. 2014. Using Models at Runtime to Address Assurance for Self-Adaptive Systems. *Models@ run. time.* Springer.

CHENG, B. H., ROG, LEMOS, R., GIESE, H., INVERARDI, P., MAGEE, J., ANDERSSON, J., BECKER, B., BENCOMO, N., BRUN, Y., CUKIC, B., SERUGENDO, G. M., DUSTDAR, S., FINKELSTEIN, A., GACEK, C., GEIHS, K., GRASSI, V., KARSAI, G., KIENLE, H. M., KRAMER, J., LITOIU, M., MALEK, S., MIRANDOLA, R., M, H. A., #252, LLER, PARK, S., SHAW, M., TICHY, M., TIVOLI, M., WEYNS, D. & WHITTLE, J. 2009. Software Engineering for Self-Adaptive Systems: A Research Roadmap. *In:* BETTY, H. C., ROG, RIO, L., HOLGER, G., PAOLA, I. & JEFF, M. (eds.) *Software Engineering for Self-Adaptive Systems.* Springer-Verlag.

CHEVERST, K., DAVIES, N., MITCHELL, K. & FRIDAY, A. 2000. Experiences of developing and deploying a context-aware tourist guide: the GUIDE project. *Proceedings of the 6th annual international conference on Mobile computing and networking.* Boston, Massachusetts, United States: ACM.

COULOURIS, G. F. 2012. *Distributed systems : concepts and design,* Boston, Mass. ; London, Addison-Wesley.

CRAWSHAW, J. & CHAMBERS, J. 2001. *A concise course in advanced level statistics: with worked examples*, Nelson Thornes.

CURBERA, F. 2007. Component Contracts in Service-Oriented Architectures. *Computer,* 40**,** 74-80.

DAMGAARD, T. C. & BIRKEDAL, L. 2006. Axiomatizing binding bigraphs. *Nordic J. of Computing,* 13**,** 58-77.

DEBOIS, S. & DAMGAARD, T. C. 2005. Bigraphs by example. *Technical Report TR-2005-61.* IT University of Copenhagen.

DESHPANDE, A., GOLLU, A. & SEMENZATO, L. 1998. The SHIFT programming language for dynamic networks of hybrid automata. *Automatic Control, IEEE Transactions on,* 43**,** 584-587.

DOWLING, J. & CAHILL, V. 2001. The k-component architecture meta-model for self-adaptive software. *Metalevel Architectures and Separation of Crosscutting Concerns***,** 81-88.

ELKHODARY, A., MALEK, S. & ESFAHANI, N. 2009. On the Role of Features in Analyzing the Architecture of Self-Adaptive Software Systems. 4th Workshop on Models@ run. time, MODELS, 2009.

ELSBORG, E. 2009. *Bigraphs:Modelling, Simulation, andType Systems : On Bigraphs for Ubiquitous Computing and on Bigraphical Type Systems.* Doctor of Philosophy, IT University of Copenhagen.

FERRY, N., HOURDIN, V., LAVIROTTE, S., REY, G., TIGLI, J. Y. & RIVEILL, M. 2009. Models at Runtime: Service for Device Composition and Adaptation.

FLOCH, J., HALLSTEINSEN, S., STAV, E., ELIASSEN, F., LUND, K. & GJORVEN, E. 2006. Using architecture models for runtime adaptability. *Software, IEEE,* 23**,** 62-70.

FRANCE, R. & RUMPE, B. 2007. Model-driven Development of Complex Software: A Research Roadmap. *2007 Future of Software Engineering.* IEEE Computer Society.

FREDJ, M., GEORGANTAS, N. & ISSARNY, V. 2006. Adaptation to Connectivity Loss in Pervasive Computing Environments. *Proceedings of the 4th MiNEMA Workshop,2006*.

GALATI, A., DJEMAME, K. & GREENHALGH, C. 2013. A mobility model for shopping mall environments founded on real traces. *Networking Science,* 2**,** 1-11.

GARDNER, M. 1970. Mathematical games: The fantastic combinations of John Conway's new solitaire game "life". *Scientific American,* 223**,** 120-123.

GARLAN, D., CHENG, S. W., HUANG, A. C., SCHMERL, B. & STEENKISTE, P. 2004. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer,* 37**,** 46-54.

GARLAN, D. & SCHMERL, B. 2004. Using architectural models at runtime: Research challenges. *Software Architecture***,** 200-205.

GASEVIC, D. 2006. *Model driven architecture and ontology development,* New York, Springer.

GAŠEVIĆ, D., DJURIĆ, D., DEVEDZIC, V. & GAŠEVIĆ, D. 2009. *Model driven engineering and ontology development,* Dordrecht ; New York, Springer.

GELERNTER, D. 1985. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.,* 7**,** 80-112.

GEORGAS, J. C., VAN DER HOEK, A. & TAYLOR, R. N. 2009. Using architectural models to manage and visualize runtime adaptation. *Computer,* 42**,** 52-60.

GHEZZI, C., JAZAYERI, M. & MANDRIOLI, D. 2002. *Fundamentals of software engineering*, Prentice Hall PTR.

GILAT, A. 2009. *MATLAB: An introduction with Applications*, John Wiley & Sons.

GJERLUFSEN, T., INGSTRUP, M. & OLSEN, J. W. 2009. Mirrors of meaning: Supporting inspectable runtime models. *Computer,* 42**,** 61-68.

GREENHALGH, C. 2009a. *Bigraph Cookbook: modelling with bigraphs* [Online]. Available: http://bigraphspace.svn.sourceforge.net/viewvc/bigraphspace/bigraphspace/docs/Bigraph_Cookbook.html [Accessed 15 August 2012].

GREENHALGH, C. 2009b. *Bigraphspace Tutorial* [Online]. Available: http://bigraphspace.svn.sourceforge.net/viewvc/bigraphspace/bigraphspace/docs/Tutorial.html [Accessed 15 August 2012].

GREENHALGH, C., LOGAN, B. & MADDEN, N. 2009. Bigraphspace: authoring ubiquitous experiences with bigraphs. *Workshop on Formal Approaches to Ubiquitous Systems (FAUSt 2009).* London.

HELAL, S. 2010. *The Landscape of Pervasive Computing Standards*, Morgan and Claypool Publishers.

HELLERSTEIN, J., PAREKH, S., DIAO, Y. & TILBURY, D. M. 2004. *Feedback control of computing systems*, Wiley-IEEE Press.

HENSON, M., DOOLEY, J., AL GHAMDI, A. A. M. & WHITTINGTON, L. 2012.Towards Simple and Effective Formal Methods for Intelligent Environments. Intelligent Environments (IE), 2012 8th International Conference on, 2012. IEEE, 251-258.

HIRSCH, D., KRAMER, J., MAGEE, J. & UCHITEL, S. 2006. Modes for software architectures. *Software Architecture,* 4344**,** 113-126.

HØJSGAARD, E. 2011. *Bigraphical Languages and their Simulation.* Doctor of Philosophy, IT University of Copenhagen.

HOURDIN, V., TIGLI, J. Y., LAVIROTTE, S., REY, G. & RIVEILL, M. 2008. SLCA, composite services for ubiquitous computing. Proceedings of the International Conference on Mobile Technology, Applications, and Systems, 2008. ACM, 11.

HUAI-GUANG, W., GUO-QING, W. & LI, W. 2010. Bigraphical model of service composition in ubiquitous computing environments. Environmental Science and Information Application Technology (ESIAT), 2010 International Conference on, 2010. IEEE, 658-662.

ITU. 2007a. *BPL Tool Architecture* [Online]. Available: http://www.itu.dk/research/pls/wiki/index.php/BPL_Tool_Architecture [Accessed 9th August 2012].

ITU. 2007b. *ML implementation of BPL*

[Online]. Available: http://www.itu.dk/research/theory/bpl/doc/ml/doc/kernel/ [Accessed 9th August

2012].

ITU. 2008. *A Brief Introduction To Bigraphs* [Online]. Available: http://www.itu.dk/research/pls/wiki/index.php/A_Brief_Introduction_To_Bigraphs [Accessed 15 September 2012].

ITU. 2011. *BPL Tool for Developers* [Online]. Available: http://www.itu.dk/research/pls/wiki/index.php/BPL_Tool_For_developers

[Accessed 9th August 2012].

JAIN, R. 1991. *The art of computer systems performance analysis*, John Wiley & Sons New York.

JENSEN, O. H. 2006. Mobile processes in bigraphs. *Unpublished monograph. http://www. cl. cam. ac. uk/rm135/Jensen-monograph. pdf (October 2006)*.

JENSEN, O. H. & MILNER, R. 2003. Bigraphs and transitions. *Acm Sigplan Notices,* 38**,** 38-49.

JENSEN, O. H. & MILNER, R. 2004. Bigraphs and Mobile Processes. *Technical Report UCAM-CL-TR-580*

. Cambridge: University of Cambridge – Computer Laboratory.

JGRAPHT. 2014. *JGraphT* [Online]. Available: http://jgrapht.org/ [Accessed July 2014].

JOHNSON, M. 1987. *The body in the mind: The bodily basis of meaning, imagination, and reason*, University of Chicago Press.

KESHAV, S. 2012. *Mathematical Foundations of Computer Networking*, Addison-Wesley Professional.

KRAMER, J. & MAGEE, J. 2007. Self-Managed Systems: an Architectural Challenge. *2007 Future of Software Engineering.* IEEE Computer Society.

LEE, W.-M. 2013. *Android Application Development Cookbook* [Online]. John Wiley & Sons, Inc. Available: http://www.wrox.com/WileyCDA/WroxTitle/Android-Application-Development-Cookbook-93-Recipes-for-Building-Winning-Apps.productCd-1118177673,descCd-DOWNLOAD.html [Accessed 15 January 2014].

LEMOS, R. D., GIESE, H., MÜLLER, H. A., SHAW, M., ANDERSSON, J., BARESI, L., BECKER, B., BENCOMO, N., BRUN, Y., CUKIC, B., DESMARAIS, R., DUSTDAR, S., ENGELS, G., GEIHS, K., GOESCHKA, K. M., GORLA, A., GRASSI, V., INVERARDI, P., KARSAI, G., KRAMER, J., LITOIU, M., LOPES, A., MAGEE, J., MALEK, S., MANKOVSKII, S., MIRANDOLA, R., MYLOPOULOS, J., NIERSTRASZ, O., PEZZÈ, M., PREHOFER, C., SCHÄFER, W., SCHLICHTING, R., SCHMERL, B., SMITH, D. B., SOUSA, J. P., TAMURA, G., TAHVILDARI, L., VILLEGAS, N. M., VOGEL, T., WEYNS, D., WONG, K. & WUTTKE, J. 2012. Software engineering for self-adaptive systems: A second research roadmap. *In:* LEMOS, R. D., GIESE, H., MÜLLER, H. A. & SHAW, M. (eds.) *Software Engineering for Self-Adaptive Systems II.* Springer-Verlag.

LISKOV, B. & GUTTAG, J. 2000. *Program development in JAVA: abstraction, specification, and object-oriented design*, Pearson Education.

MAGEE, J., DULAY, N., EISENBACH, S. & KRAMER, J. 1995. Specifying Distributed Software Architectures. *Proceedings of the 5th European Software Engineering Conference.* Springer-Verlag.

MARINO, J. & ROWLEY, M. 2010. *Understanding SCA (Service Component Architecture),* Upper Saddle River, NJ, Addison-Wesley.

MCKINLEY, P. K., SADJADI, S. M., KASTEN, E. P. & CHENG, B. H. C. 2004. Composing Adaptive Software. *Computer,* 37**,** 56-64.

MILNER, R. 1999. *Communicating and mobile systems: the pi calculus*, Cambridge university press.

MILNER, R. 2004a. Bigraphs for Petri nets. *Lectures on Concurrency and Petri Nets***,** 161-191.

MILNER, R. 2004b. Bigraphs whose names have multiple locality. *Technical report, University of Cambridge Computer Laboratory*.

MILNER, R. 2006a. Pure bigraphs: Structure and dynamics. *Information and computation,* 204**,** 60-122.

MILNER, R. 2006b. Ubiquitous Computing: Shall we Understand It? *Comput. J.,* 49**,** 383-389.

MILNER, R. 2008a. Bigraphs and Their Algebra. *Electron. Notes Theor. Comput. Sci.,* 209**,** 5-19.

MILNER, R. 2008b. *Lecture notes on Bigraphs: a Model for Mobile Agents* [Online]. Available: http://www.cl.cam.ac.uk/archive/rm135/Bigraphs-Notes.pdf [Accessed 15th November 2012].

MILNER, R. 2009. *The space and motion of communicating agents,* Cambridge New York, Cambridge University Press.

MORIN, B., BARAIS, O., JÉZÉQUEL, J. M., FLEUREY, F. & SOLBERG, A. 2009. Models@ run. time to support dynamic adaptation. *Computer,* 42**,** 44-51.

MORIN, B., FLEUREY, F., BENCOMO, N., JEZEQUEL, J. M., SOLBERG, A., DEHLEN, V. & BLAIR, G. 2008. An Aspect-Oriented and Model-Driven Approach for Managing Dynamic Variability. *Model Driven Engineering Languages and Systems, Proceedings,* 5301**,** 782-796.

OMG. 2011. *Meta Object Facility Core Specification OMG* [Online]. Available: http://www.uml3.ru/library/uml_spec/mof.pdf [Accessed November 2014.

OREIZY, P., GORLICK, M. M., TAYLOR, R. N., HEIMHIGNER, D., JOHNSON, G., MEDVIDOVIC, N., QUILICI, A., ROSENBLUM, D. S. & WOLF, A. L. 1999. An architecture-based approach to self-adaptive software. *Intelligent Systems and Their Applications, IEEE,* 14**,** 54-62.

PAPAZOGLOU, M. P., TRAVERSO, P., DUSTDAR, S. & LEYMANN, F. 2007. Service-Oriented Computing: State of the Art and Research Challenges. *Computer,* 40**,** 38-45.

PARNAS, D. L. 2010. Really rethinking'formal methods'. *Computer,* 43**,** 28-34.

PEREIRA, E., KIRSCH, C. & SENGUPTA, R. 2012. *BiAgents-A Bigraphical Agent Model for Structure-Aware Computation* [Online]. Available: http://cpcc.berkeley.edu/papers/paperBiagents12.pdf [Accessed 1 October 2012].

PERRONE, G., DEBOIS, S. & HILDEBRANDT, T. T. 2012. A model checker for Bigraphs. Proceedings of the 27th Annual ACM Symposium on Applied Computing, 2012. ACM, 1320-1325.

POLYVYANYY, A., SMIRNOV, S. & WESKE, M. 2008. Process Model Abstraction: A Slider Approach. *Proceedings of the 2008 12th International IEEE Enterprise Distributed Object Computing Conference.* IEEE Computer Society.

POSLAD, S. 2009. *Ubiquitous Computing: Smart Devices, Environments and Interactions*, Wiley Publishing.

RAMIREZ, A. J. & CHENG, B. H. C. 2009. Evolving models at run time to address functional and non-functional adaptation requirements. 4 th Workshop on Models@ run. time at MODELS 09, 2009.

RUSSELL, S. J. & NORVIG, P. 2010. *Artificial intelligence: a modern approach*, Prentice hall Upper Saddle River, NJ.

SALTZER, J. H. & KAASHOEK, M. F. 2009. *Principles of Computer System Design: An Introduction*, Morgan Kaufmann Publishers Inc.

SCHMIDT, D. C. 2006. Guest Editor's Introduction: Model-Driven Engineering. *Computer,* 39**,** 25-31.

SEVEGNANI, M. & CALDER, M. 2010. Bigraphs with sharing. *University of Glasgow, Tech. Rep*.

SOMMERVILLE, I. 2011. *Software engineering,* Boston ; London, Pearson.

SVENTEK, J., KOLIOUSIS, A., SHARMA, O., DULAY, N., PEDIADITAKIS, D., SLOMAN, M., RODDEN, T., LODGE, T., BEDWELL, B. & GLOVER, K. 2011. An information plane architecture supporting home network management. Integrated Network Management (IM), 2011 IFIP/IEEE International Symposium on, 2011. IEEE, 1-8.

SYKES, D., HEAVEN, W., MAGEE, J. & KRAMER, J. 2008. From goals to components: a combined approach to self-management. Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems, 2008. ACM, 1-8.

ULLMAN, J. D. 1998. *Elements of ML Programming: ML97 Edition*, Prentice Hall.

WADDINGTON, D. & LARDIERI, P. 2006. Model-centric software development. *IEEE Computer,* 39**,** 28-29.

WAGNER, G., ANTONIOU, G., TABET, S. & BOLEY, H. 2004. The abstract syntax of RuleML-towards a general web rule language framework.

WALTON, L. & WORBOYS, M. 2009. An algebraic approach to image schemas for geographic space. *Spatial Information Theory***,** 357-370.

WALTON, L. & WORBOYS, M. 2012. A Qualitative Bigraph Model for Indoor Space. *Geographic Information Science***,** 226-240.

WANG, J. S., XU, D. & LEI, Z. 2011. Formalizing the Structure and Behaviour of Context-Aware Systems in Bigraphs. Software and Network Engineering (SSNE), 2011 First ACIS International Symposium on, 2011. IEEE, 89-94.

WANG, L., ZHANG, G., ZHU, J. & WU, J. 2010. A method for modeling aspect-oriented dynamic software architecture. Computer Science and Education (ICCSE), 2010 5th International Conference on, 2010. IEEE, 85-90.

WEISER, M. 1999. The computer for the 21st century. *SIGMOBILE Mob. Comput. Commun. Rev.,* 3**,** 3-11.

XU, D. Z., XU, D. & LEI, Z. 2011. Bigraphical Model of Context-Aware in Ubiquitous Computing Environments. Services Computing Conference (APSCC), 2011 IEEE Asia-Pacific, 2011. IEEE, 389-394.

XUE, G., KONG, H., LIU, X. & YAO, S. 2009. Modeling Service Interactions in Term of Bigraphs. New Trends in Information and Service Science, 2009. NISS'09. International Conference on, 2009. IEEE, 117-122.

XUE, G., ZHANG, K., YANG, J. & YAO, S. 2011. Plain abstraction of business process model. Computer Sciences and Convergence Information Technology (ICCIT), 2011 6th International Conference on, 2011. IEEE, 338-341.

ZHAI, H., ZHANG, W., CUI, L., LIU, H. & ABRAHAM, A. 2011a. A Bigraph Model for Multi-route Choice in Urban Rail Transit. Communication Systems and Network Technologies (CSNT), 2011 International Conference on, 2011a. IEEE, 699-703.

ZHAI, H., ZHANG, W., CUI, L., SHI, J. & LI, H. 2011b. Toward formal description to metro services mechanism based on bigraph models. Soft Computing and Pattern Recognition (SoCPaR), 2011 International Conference of, 2011b. IEEE, 285-289.

ZHAI, H., ZHANG, W., CUI, L., XIE, X. & ZHANG, X. 2011c. High-confidence petroleum industrial critical systems research based on bigraphical models. Soft Computing and Pattern Recognition (SoCPaR), 2011 International Conference of, 2011c. IEEE, 290-295.

ZHANG, M., SHI, L., ZHU, L., WANG, Y., FENG, L. & PU, G. 2008. A Bigraphical Model of WSBPEL. *Proceedings of the 2008 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering.* IEEE Computer Society.