Mobile Reactive Systems over Bigraphical Machines - A Programming Model and its Implementation

by

Elói Teixeira Pereira

A dissertation submitted in partial satisfaction of the requirements for the degree of
Doctor of Philosophy

in

Engineering - Civil and Environmental Engineering

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Raja Sengupta, Chair
Professor Edward Lee
Professor Alexandre Bayen

Summer 2015
Mobile Reactive Systems over Bigraphical Machines - A Programming Model and its Implementation

Copyright 2015
by
Elói Teixeira Pereira
Abstract

Mobile Reactive Systems over Bigraphical Machines - A Programming Model and its Implementation

by

Elói Teixeira Pereira

Doctor of Philosophy in Engineering - Civil and Environmental Engineering

University of California, Berkeley

Professor Raja Sengupta, Chair

In this dissertation we address the problem of bridging reactive programs and mobile computing machinery embedded in physical spaces with dynamic structure. We propose the BigActor Model as a bridging model between programs and logical-space models. The BigActor model [1] combines Hewitt and Agha's Actor model [2] for specifying concurrent reactive programs with Robin Milner's Bigraphical Model [3] for specifying the location and connectivity of the computing machines. The BigActor Model makes location and connectivity first-class citizens in distributed machines. This is analogous to another bridging model, the von Neumann machine, which makes first-class citizens of memory, instructions, and their sequentiality. The BigActor Programming Language (BAL) is an implementation of the BigActor Model. It has a runtime system named the BigActor Runtime System (BARS). The BARS targets an abstract machine (bigraphs). The abstract machine has to be realized on a physical space of mobile and distributed computing machines. The realization is produced by the Logical-Space Execution Engine (LSEE), which bridges bigraphs with the physical space. The Logical-Space Runtime System (LSRS) extends BARS with LSEE so that programs written in BAL can seamlessly execute over physical spaces.

The second part of this dissertation is concerned with the formalization and implementation of the interactions between logical spaces and physical spaces. First, we approach this problem formally, by introducing the logical-space computing semantics. In logical-space computing, spatial agents operate over logical-space models while the runtime system is in charge of interacting with the physical space. We presented an implementation that follows the logical-space computing semantics. The LSRS uses the LSEE to generate logical-space models using bigraphs. The physical space is modelled using polygons defined using GPS coordinates. The spatial agents are bigActors. Our implementation programs robots and sensors in logical-space to execute an oil-spill monitoring exercise in the Atlantic. BigActor programs execute over BARS, which interacts with physical spaces through the LSEE. LSEE executes over the Robot Operating System (ROS) - an open-source middleware for robotics. The physical machinery used in the demonstration consisted of one Air Force UAV, three
ground control stations, four drifters that broadcast their position using AIS, and one Navy vessel equipped with a small speedboat. The Portuguese Navy emulated the oil-spill by releasing 100kg of popcorn in the ocean.
To my parents Lourdes and Artur
# Contents

Contents

List of Figures iv

List of Tables vii

1 Introduction 1

1.1 Models bridging programs and machines 4

1.2 Spatial Models 5

1.3 Spatial programming models 7

I Reactive programming on bigraphical machines 10

2 Observing and Controlling Bigraphs 11

2.1 Bigraphical formalism - a review 11

2.2 Querying for local observations 20

2.3 Local bigraphical reaction rules 24

2.4 Final remarks 25

3 BigActors 27

3.1 Introduction 27

3.2 Actor model of computation - review 29

3.3 BigActor Model 34

3.4 BigActors Operational semantics 38

3.5 Communication back-channelling 50

3.6 Correctness of BigActor Model Semantics 52

3.7 Final remarks 58

4 BigActor Programming Language 59

4.1 What is an Embedded DSL 60

4.2 Scala Programming Language 60

4.3 BARS - The BigActor Runtime System 63
4.4 A simulation environment for BigActors ............................................ 68  
4.5 A case study in mobile robotics .......................................................... 71  
4.6 Final remarks ....................................................................................... 77  

II Bridging logical and physical machines .................................................. 80  

5 Logical-Space Computing ......................................................................... 81  
5.1 Introduction ......................................................................................... 81  
5.2 Spatial Structures .............................................................................. 84  
5.3 Bigraphs and Polygons ...................................................................... 85  
5.4 Consistency of spatial structures ......................................................... 89  
5.5 Logical-Space Computing Semantics .................................................. 91  
5.6 Correctness of control actions ............................................................. 95  
5.7 Using the semantics to enforce local consistency ................................. 99  
5.8 Final remarks ....................................................................................... 102  

6 Case study: oil-spill monitoring mission .................................................. 104  
6.1 The bilge dumping problem ................................................................. 104  
6.2 Vehicles and Sensors ......................................................................... 109  
6.3 Logical-Space Runtime System ............................................................. 110  
6.4 Logical-Space Execution Engine .......................................................... 111  
6.5 Final remarks ....................................................................................... 119  

7 Conclusions ............................................................................................. 121  

Bibliography ............................................................................................... 123
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>A bridging model between programs and machines with dynamic structure.</td>
<td>2</td>
</tr>
<tr>
<td>2.1</td>
<td>Example of a bigraph.</td>
<td>12</td>
</tr>
<tr>
<td>2.2</td>
<td>Placing and Linking graphs of the bigraph of Figure 2.1.</td>
<td>12</td>
</tr>
<tr>
<td>2.3</td>
<td>Example of a bigraph from $\mathcal{B}g(\mathcal{K}_{city})$.</td>
<td>15</td>
</tr>
<tr>
<td>2.4</td>
<td>Composition of the bigraph $streetMap$ with the bigraph $networkInf$.</td>
<td>17</td>
</tr>
<tr>
<td>2.5</td>
<td>Abstract BBRs $MOVE$ that moves a $Smart$ node from its current location to a $Street$ node, and $CONNECT$ that connects the $Smart$ node to a $Wlan$ node.</td>
<td>18</td>
</tr>
<tr>
<td>2.6</td>
<td>Example of the application of the $MOVE$ reaction rule.</td>
<td>19</td>
</tr>
<tr>
<td>2.7</td>
<td>A bigraph trace resulting from applying the BRRs $MOVE(sp,street1)$, $MOVE(sp,street3)$, and $CONNECT(sp,wlan0)$.</td>
<td>20</td>
</tr>
<tr>
<td>2.8</td>
<td>BigActor observing the street map using query $CHILDREN(PARENT(HOST))$.</td>
<td>21</td>
</tr>
<tr>
<td>3.1</td>
<td>A pictorial representation of a bigActor system.</td>
<td>28</td>
</tr>
<tr>
<td>3.2</td>
<td>Specification of the bigActor system for Example 3.1.</td>
<td>29</td>
</tr>
<tr>
<td>3.3</td>
<td>Execution trace for the Example 3.1.</td>
<td>30</td>
</tr>
<tr>
<td>3.4</td>
<td>The Actor Model.</td>
<td>31</td>
</tr>
<tr>
<td>3.5</td>
<td>Grammar for actors syntax.</td>
<td>32</td>
</tr>
<tr>
<td>3.6</td>
<td>Actor operational semantics.</td>
<td>32</td>
</tr>
<tr>
<td>3.7</td>
<td>$app$ and $social$ actors.</td>
<td>34</td>
</tr>
<tr>
<td>3.8</td>
<td>Execution trace of the actor system ${app, social}$.</td>
<td>35</td>
</tr>
<tr>
<td>3.9</td>
<td>Actor system for Example 3.2 with new actors $app$, $smartphone$, and $env$.</td>
<td>36</td>
</tr>
<tr>
<td>3.10</td>
<td>BigActor system for scenario presented in Example 3.3.</td>
<td>37</td>
</tr>
<tr>
<td>3.11</td>
<td>BRRs $MOVE_HOST_TO(Loc)$ and $CONNECT_HOST_TO(Wlan)$.</td>
<td>38</td>
</tr>
<tr>
<td>3.12</td>
<td>Grammar for bigActors syntax.</td>
<td>39</td>
</tr>
<tr>
<td>3.13</td>
<td>BigActors operational semantics.</td>
<td>40</td>
</tr>
<tr>
<td>3.14</td>
<td>BigActor system for scenario presented in Example 3.5.</td>
<td>42</td>
</tr>
<tr>
<td>3.15</td>
<td>BigActors of Figure 3.14 embedded in the bigraph of Figure 2.3.</td>
<td>42</td>
</tr>
<tr>
<td>3.16</td>
<td>Execution trace for the bigActor system of Figure 3.14.</td>
<td>44</td>
</tr>
<tr>
<td>3.17</td>
<td>Execution trace over the respective Bigraph Reactive System.</td>
<td>45</td>
</tr>
<tr>
<td>3.18</td>
<td>Actor system synthesized to produce the projected trace of Figure 3.16.</td>
<td>49</td>
</tr>
<tr>
<td>3.19</td>
<td>Areas of influence $A_{app2}^{B_0}$ and $A_{\text{social}_2}^{B_0}$.</td>
<td>51</td>
</tr>
</tbody>
</table>
3.20 BigActor grabberBA ................................................................. 53
3.21 Example of an execution of grabberBA leading to an unsafe configuration. ... 53
3.22 BigActors grabberBA0 and grabberBA1 ........................................ 54
3.23 Example of an execution of multiple bigActors leading to an unsafe configuration. 54

4.1 A simple actor class and its instantiation and invocation .......................... 61
4.2 A simple actor instantiated and invoked using the actor method ................ 62
4.3 Remote actor instantiation .......................................................... 62
4.4 Remote actor selection and use ...................................................... 62
4.5 BigActor Runtime System ............................................................ 63
4.6 Definition of BigActor class .......................................................... 64
4.7 Instantiation and invocation of a BigActor ......................................... 64
4.8 Instantiation and invocation of a BigActor ......................................... 65
4.9 Algebraic data type BigActorSchdlAPI that defines a set of messages to interact with BigActorSchdl .................................................. 66
4.10 Code skeleton for BigActorSchdl .................................................... 66
4.11 Algebraic data type BigraphManagerAPI that defines a set of messages to interact with BigraphManager .......................................... 67
4.12 Remote bigActor registering, selection and use .................................... 68
4.13 Remote bigActor instantiation ...................................................... 68
4.14 BGM grammar ............................................................................ 69
4.15 Term specifying the bigraph of Figure 2.3 .......................................... 69
4.16 BigActor Runtime Syste on BigMC .................................................. 70
4.17 Example of a bigraph generation for the mobile robotics case study ............ 72
4.18 BGM term for the bigraph depicted in Figure 4.17 ................................. 73
4.19 Abstract BRRs modelling the set of control actions ............................... 74
4.20 Abstract BRRs modelling the set of environment actions ......................... 74
4.21 BigActor specifying the oil-spill monitoring mission ............................... 75
4.22 Definition of track_oilSpills .......................................................... 76
4.23 Definition of sample_spill .............................................................. 76
4.24 Definition of ais_receiver ................................................................ 77
4.25 Timeline for an execution of the BigActor system that specifies the oil-spill monitoring example .................................................. 78
4.26 Definition of broadcast command that broadcasts a message to all bigActors hosted at nodes linked to the host of the bigActors executing the command .................................................. 79

5.1 Physical-space execution for Example 5.1 .............................................. 82
5.2 Symbolical-space execution for Example 5.1 ......................................... 83
5.3 BRRs that generates the symbolic-space execution depicted in Figure 5.2 ...... 83
5.4 Logical space and physical space ....................................................... 86
5.5 Logical-space execution for Example 5.1 .............................................. 87
5.6 Logical-space execution for Example 5.5 .............................................. 89
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.7</td>
<td>Inconsistent structure.</td>
</tr>
<tr>
<td>5.8</td>
<td>Spatial agent modelled as a bigActor.</td>
</tr>
<tr>
<td>5.9</td>
<td>Logical-space execution for the spatial agent of Figure 5.8.</td>
</tr>
<tr>
<td>5.10</td>
<td>Example of a feedback specified as a bigActor.</td>
</tr>
<tr>
<td>6.1</td>
<td>CleanSeaNet SAR image with evidences of bilge dumping acquired in June 2009 off the coast of Spain. The suspected vessel was also detected in the SAR image. The satellite image is compared with SafeSeaNet AIS database in order to identify the suspected ship. Source: EOMag [4].</td>
</tr>
<tr>
<td>6.2</td>
<td>High-resolution satellite image from the oil spill. Courtesy of EMSA.</td>
</tr>
<tr>
<td>6.3</td>
<td>Alfa Extended UAV.</td>
</tr>
<tr>
<td>6.4</td>
<td>Onboard UAV picture of the oil-spill (popcorn) and Navy vessel.</td>
</tr>
<tr>
<td>6.5</td>
<td>Mission Visualization Tool.</td>
</tr>
<tr>
<td>6.6</td>
<td>BigActor Runtime System for Logical-Space Programming.</td>
</tr>
<tr>
<td>6.7</td>
<td>ROS message definition for MobilityCommand.</td>
</tr>
<tr>
<td>6.8</td>
<td>ROS message definition for PhysicalProp.</td>
</tr>
<tr>
<td>6.9</td>
<td>ROS message definition for Location.</td>
</tr>
<tr>
<td>6.10</td>
<td>ROS message definition for Connection.</td>
</tr>
<tr>
<td>6.11</td>
<td>ROS message definition for Bigraph.</td>
</tr>
<tr>
<td>6.12</td>
<td>Bigraph Driver example.</td>
</tr>
<tr>
<td>6.13</td>
<td>Distributed bigraph example.</td>
</tr>
<tr>
<td>6.14</td>
<td>BRR Driver example.</td>
</tr>
<tr>
<td>6.15</td>
<td>Logical-space program for observing and tracking the oil-spill.</td>
</tr>
<tr>
<td>6.16</td>
<td>Code for bigActor handover.</td>
</tr>
</tbody>
</table>
List of Tables

3.1 Encoding of Scala actor commands into the actors grammar. . . . . . . . . . . . 33
4.1 Sorting discipline for the mobile robotics case study. . . . . . . . . . . . . . . . 72
5.1 Physical interpretation for the logical-space execution depicted by Figure 5.5 . . 88
Acknowledgments

First and foremost I want to thank my adviser, Professor Raja Sengupta, for all his guidance and support. Our collaboration started in 2007 when I did an internship at the Center for Collaborative Control for Unmanned Vehicles (C3UV). Professor Sengupta made an assessment of my research capabilities and limitations and put me in an unorthodox research path - exploring the boundaries between the areas of “programming models” and “robotics”. This dissertation is the result of our research endeavour, which was built from endless meetings brainstorming on the white board or on a sheet of paper.

I would like to thank my dissertation committee, Professor Edward Lee and Professor Alexandre Bayen for reading my dissertation, providing invaluable feedback. Professor Lee has been endorsing this research work from the day I had the pleasure to present it to the Ptolemy II research group. He provided insightful feedback regarding the formalization of the model of computation and its applicability to swarm systems.

I want to thank the financial support from the “Fundaç~ao para a Ciencia e Tecnologia” and the Portuguese Studies Program (PSP) / Pinto-Fialon fellowship. In particular I want thank the director of the PSP, Professor Deolinda Ad~ao, for all her support and help since the day I decided to apply to UC Berkeley.

I want to thank Professor Christoph Kirsch from University of Salzburg. Professor Kirsch participated in the discussions that spawned our main research ideas. His expertise in the area of “programming languages” was of paramount importance for the success of this work.

I want to thank Professor Bob Iannucci from Carnegie Mellon University for reading and commenting this dissertation. Professor Iannucci attended a presentation where I introduced our model of computation at a very early stage.

I want to thank Professor Jo˜ao Sousa from Porto University for his advise at early stages of my doctoral studies and for his research suggestions.

I want to thank the Civil and Environmental Engineering Department, in particular the graduate student adviser Shelley Okimoto.

I want to thank the Principal Investigators of the Cyber-Physical Cloud Computing lab (CPCC), Professor Sengupta and Professor Kirsch, and the chief of the Portuguese Air Force Academy Research group (CIAFA), Colonel Jos´e Morgado.

I would also like to thank my colleagues at CPCC and CIAFA, in particular the help and support of Joshua Love, Hao Chen, Robert Hansen, Tiago Oliveira, Gonçalo Cruz and Francisco Machado. I would like to give a special thanks for Pedro Marques da Silva and Clemens Krainer for their tremendous influence and contributions on the software implementation.

I would like to thank the Ptolemy II research group, in particular Professor David Broamn and Professor Stavros Tripakis for the help on the formalization of the semantics of the model.

Throughout the times of this PhD I had the pleasure to make some amazing friends. I am thankful for the support and friendship of Maria, Catarina, Gonçalo, Osní, Ricardo Sousa, Blanca, Tim, Aude, Ryan, Samitha, Jack, Yusef, Cavaco, and Ricardo Guerreiro.

Now it is time for my Berkeley family. I would like to thank Ana. We started our PhD journeys together and it is amazing to realize what we accomplished. I want to thank
Richard and Sylvia for their support and advice provided during sweet breakfast chats or during inspirational hikes. I want to thank the Abreu’s family for always making me feel part of their own family. I am tremendously thankful to my “irmãos da vida” Carlos Oliveira and Luís dos Santos for their companionship, and to my “greek girl” Eleftheria Stavridi for her unconditional support and for keeping my levels of sanity, confidence, and motivation high so that I could reach the end of this dissertation.

Now in Portuguese. Quero agradecer a toda a minha família em Portugal, em França e aos restantes emigrados por este mundo. Agradeço os meus avós Margarida, Adriano, Rosalina e Antonio que muito lutaram para dar um futuro as suas famílias. Por fim, quero dedicar esta dissertação aos meus pais, Lourdes e Artur. As suas vidas são histórias de perseverança e determinação. São as minhas principais fontes de motivação e de exemplo. Estou tremendamente grato pelo vosso apoio incondicional, amor e educação. Bem hajam!
Chapter 1

Introduction

In this dissertation we address the problem of bridging reactive programs and mobile computing machinery embedded in physical spaces with dynamic structure. Our approach is two-fold. In the first part of we propose the BigActor Model as a bridging model between programs and bigraphical machines. The BigActor model [1] combines Hewitt and Agha’s Actor model [2] for specifying concurrent reactive programs with Robin Milner’s Bigraphical Model [3] for specifying the location and connectivity of the computing machines. The BigActor Model makes location and connectivity first-class citizens in distributed and mobile systems. The approach is motivated by another bridging model, the von Neumann machine, which makes first-class citizens of memory, instructions, and their sequentiality.

The BigActor Programming Language (BAL) is an implementation of the BigActor Model. It has a runtime system named the BigActor Runtime System (BARS). The BARS targets an abstract machine modelled as bigraphs. The abstract machine must be realized on a physical-space of mobile and distributed computing machines. The second part of this dissertation is concerned with the formalization and implementation of this realization.

We propose a semantics named Logical-Space Computing, where spatial agents operate over logical-space models, such as bigraphs, while the runtime system is in charge of interacting with the physical space. The mediation of these interactions is performed by the Logical-Space Execution Engine (LSEE). The Logical-Space Runtime System (LSRS) extends BARS with LSEE so that programs written in BAL can seamlessly execute over physical spaces. Our implementation of the LSEE uses bigraphs as logical-space models and polygons defined using GPS coordinates as physical-space models. Spatial agents are modeled as bigActors. Our implementation provides means for logical-space programming of autonomous vehicles and a variety of spatial sensors. We present a case study where Unmanned Aerial Vehicles (UAVs), GPS drifters, and vessels are programmed in logical-space to perform a oil-spill monitoring mission.

Figure 1.1 depicts our approach for bridging logical-space programs and the physical machine of networked autonomous vehicles and sensors. The bottom of Figure 1.1 depicts the physical layer composed of heterogeneous computing machinery such as Unmanned Aerial Vehicles (UAVs), vessels, ground control stations, drifters, and submarines. In the middle
CHAPTER 1. INTRODUCTION

Figure 1.1: A bridging model between programs and machines with dynamic structure.

we depict the bigraphical abstraction. The bigraph models location and connectivity of the physical machinery, e.g., uav1 is located at oilSpill0 and is connected to gcs1. The top of Figure 1.1 shows bigActor programs hosted at bigraph nodes, e.g., the first bigActor is hosted at uav0 and its behavior consists of moving its host to the location oilSpill0. Note that programs are written over a symbolic model of space, i.e., a bigraph. The interactions with the physical-space are specified by the logical-space computing semantics.

Computation is facing a paradigm shift from algorithmic calculation to interaction [5, 6]. The logic foundations of computation created by Alan Turing assumes that a computer is a machine that performs calculations in an algorithmic fashion. This assumption is carried by John von Neumann in the definition of the von Neumann machine [7]. According to von Neumann, computing systems are devices that “carry out instructions to perform calculations of a considerable order of complexity, e.g., to solve a non-linear partial differential equation” [7]. Valiant [8] claims that the von Neumann model is of great engineering value since it is the bridge that enables the chaotic world of sequential computer programs to run efficiently over the chaotic world of hardware. The von Neumann model became the de facto computing machine abstraction, which was followed by hardware industry making the computer a consumer product.

Nonetheless, performing pure calculations such as solving partial differential equations does not resonate for computation running over modern computing devices such as smartphones, autonomous vehicles, robots, and wireless sensor networks. Modern computation is
CHAPTER 1. INTRODUCTION

Chapter 1: Introduction

distributed, mobile, reactive, and interactive [9]. Computation lives ubiquitously embedded and spatially distributed over the physical space. Computation moves together with computing machines. Computation reacts upon information from human users, and interacts with the environment by means of sensors and actuators that equip modern computing machinery.

Perhaps the most significant conceptual difference between modern computing machines and the ones envisioned by Turing and von Neumann is that modern computing machines change their structure over time. The environment is now explicitly part of computing machines and its dynamics can not be overlooked. For example, smartphones and autonomous vehicles have the ability to move from one location to another, while their communication structure may change deliberately or by influence of the environment. Computation executing over these devices may execute differently according to their spatial context, e.g., requesting a smartphone to connect over a different network. Computation may also influence the environment, e.g., requesting an autonomous vehicle to change its location. In their positioning paper, Zambonelli and Mamei [10, 11] advocate that modern computing machinery needs new models of computation with spatial abstractions. Programming models must include, as first-class citizens, means for computation to locally sense the environment, to actuate over the environment, and to express mobility.

Computation that exhibits spatial behavior is commonly known as spatial computing [12]. Spatial programming languages provide programmers with spatial constructs as a first-class citizen. Just as the von Neumann model became the de facto bridging model for sequential computation, one needs new bridging models to connect the world of spatial programs and the emergent world of mobile computing devices, robots, and autonomous vehicles. We address this problem from a theoretical point of view with the BigActor model and the logical-space computing semantics, and provide a practical implementation in the area of mobile robotics.

The remainder of this dissertation goes as follows. In the remaining of Chapter 1, we position our work with respect to the literature on bridging models of computation, on programming models for systems with dynamic structure, and on spatial computing models.

In Chapter 2 we review the bigraphical formalism and introduce a query language for requesting local bigraphical observations.

In Chapter 3 we introduce the semantics of the BigActor model. The semantics is formalized using an operational style. We also formalize the correctness of BigActor executions and provide sufficient conditions to achieve it. In Chapter 4 we present an implementation of the model. The model is implemented as a Scala embedded Domain-Specific Language, called the BigActor Language (BAL). BAL programs runs over the BigActor Runtime System (BARS), which interfaces with a bigraphical abstraction of the world. The bigraphical executions are handled by a bigraph model checker. This finishes Part I of the thesis.

Part II of the thesis is concerned with bridging logical spatial abstractions with the physical world. This is called logical-space computing. In Chapter 5 we introduce the logical-space computing semantics and present a case where logical spaces are specified as bigraphs and physical spaces are specified using geometrically-defined polygons. We demonstrate the use of bigActors as a logical-space programming language. In Chapter 6 we present a case
study of using logical-space computing to program vehicles and sensors performing an oil-spill monitoring scenario. Our approach is to use bigraphs as a logical model of vehicles, sensors and their surrounding maritime environment and use bigActors as a programming model for specifying their interactive reactive behaviour. Logical-space programs are specified using bigActors. We build a software system to explore and demonstrate the value of logical-space computing when integrating and controlling a network of robots sensing an environment. The physical model is implemented over the Robot Operating System (ROS) - an open-source middleware for robotics. The physical machinery used in the demonstration consisted in one Air Force UAV, three ground control stations, four drifters that broadcast their position using AIS, and one Navy vessel equipped with a small speedboat. The Portuguese Navy emulated the oil-spill by releasing 100kg of popcorn in the ocean. We finish this dissertation in Chapter 7 with conclusions and future work.

1.1 Models bridging programs and machines

The von Neumann Machine (VNM) is a computer architecture for sequential programs [7]. As Valiant stated, VNM provides a bridge between sequential programs and the rich world of computer hardware [8]. VNM was designed for stand-alone computation and, as such, it does not entail means for explicitly model distributed systems.

Valiant [8] addresses this problem by introducing the Bulk-Synchronous Parallel (BSP) model for bridging computation and distributed computing systems. Nonetheless, BSP does not provide constructs for modelling mobility of computing devices and dynamic topology of the networks that connect them.

One can find in the literature programming models that provide dynamic structure as first-class citizen. SHIFT [13] is a language that implements Dynamic Networks of Hybrid Automata (DNHA) where components modelled as hybrid systems can be created, interconnected and destroyed as the execution evolves. R-Charon [14] extends Charon, an agent-based language for specifying interacting hybrid systems, with network reconfiguration. Influenced by SHIFT and R-Charon, we introduced the Structure Model (SM) [15] for modelling and controlling the structure of networked robotic systems. The model introduces structural dynamics to Nancy Lynch’s Synchronous Network Model [16]. The actor model [2] was introduced by Hewitt and Agha as a model for reactive, concurrent, and distributed systems. An actor can communicate with another actor if it has its address. Actor systems can implicitly model dynamic structure, evidenced by the communication of addresses between different actors. With the $\pi$-calculus [17], Milner addresses the problem of dynamic connectivity explicitly by defining channels as first-class citizens. Luca Cardelli took a different route by pointing that computation is not only constrained by its communication topology but also by its location. Cardelli’s Ambient Calculus [18] addresses the problem of mobility by defining ambients - a bounded location where computation may occur. Influenced by Hewitt’s and Cardelli’s work, Robin Milner introduced Bigraphs, combining both dynamic location and connectivity in a single model [3].
Debois and Milner posed the problem of how to bridge Turing-computation and ubiquitous computing machines with complex physical behaviors [19]. The authors claim that, in ubiquitous systems, the complex world of machines and their environment demand different models than the ones used to describe Turing-computation. Their approach is to model both layers using two different Bigraph Reactive Systems that are semantically combined.

In his essay “The tower of informatic models” [20], Robin Milner introduces a modelling practice to semantically combine models that address different aspects of computing behaviour. In a tower of models, one can combine models horizontally, e.g., two models with two different semantics are combined together to form a third one. One can also vertically explain a model into another one, as far as the target model is at least as expressive as the original one. This practice is analogous to compilation.

Influenced by Milner’s tower of models, we introduced the BigActor model for bridging computation and mobile systems with dynamic structure [1, 21]. In [19], Debois and Milner use only Bigraph Reactive Systems as modelling formalism. We take a different approach. We model computation using the Actor model [2] - a well known model of concurrent computation adopted by large spectrum of modern programming languages. For modelling machines and their dynamic structural behavior we use Bigraph Reactive Systems. The two models are horizontally combined by the BigActor semantics. The first step towards a vertical explanation of bigActors is presented in [22], where we introduce a bigraphical encoding of actors.

The design decisions behind the BigActor model relied on the following engineering considerations. The asynchronous-message passing provided by the actor semantics suits the networked robotic applications that we primarily target. Moreover, the actor model has been adopted as the concurrent model of a large number of modern programming languages such as Scala [23], Erlang [24], Ptolemy II [25], and the Akka framework [26]. We are interested in robotic systems that vary their location and connectivity during their execution, e.g., an UAV changing its communication capabilities while flying from one location to another. Since both location and connectivity are first-class citizens in bigraphs [3], we choose Milner’s model as the underlying spatial model.

1.2 Spatial Models

Location models are used in a variety of applications ranging from ubiquitous systems, autonomous vehicles, indoor robotics, and mobile computing. According to Hightower and Borriello [27] there are two classes of location models for ubiquitous computing: physical and symbolical. Physical location models are concerned with geographical position of an entity in the world. Symbolic location models define symbolically the location of entities.

Becker and Durr [28] categorize symbolic space models into set-based, graph-based models. Set-based models define space as set of names, e.g., the collection of house numbers in US Postal addresses. Using set intersection one can implicitly determine the overlapping between locations and containment relations. Graphs are used to explicitly model physi-
cal relations between sets of symbolic names. For example, vertices can be used to model physical adjacency or containment relation between names.

While physical location models are commonly used for applications that require geographical information of the world, e.g., GPS waypoint control of unmanned vehicles, symbolic models are used in domains such as indoor localization and navigation [29, 30, 31], robot exploration and mapping [32], specification and verification of robotic trajectories [33].

The bigraphical formalism introduced by Robin Milner [3] is a graph-based symbolic model that models explicitly both location and connectivity. The model uses two graphical structures over the same set of nodes, a forest named placing graph and an hypergraph named linking graph. The placing graph models containment relations, e.g., a smartphone located inside a room which is inside a building. The linking graph models connectivity between entities, e.g., physical adjacency between two rooms or network connectivity of a smartphone. With the bigraphical formalism one can also model dynamical properties of space by means of Bigraph Reaction Rules (BRR). A set of bigraphs and a set of reactive rules that models their dynamics is known as a Bigraph Reactive System (BRS). With the advent of informatics and the internet, Milner envisioned that location and connectivity are necessary for modelling mobility of ubiquitous computing systems.

The literature shows other uses of bigraphs for modelling space. Walton and Worboys [34, 35] propose the use of bigraphs for modelling indoor environments and the dynamics of entities within them. Birkedal et al. [36] present bigraphs as a model for context-aware and symbolic location-aware systems. In our case study, vehicles and sensors change both location and connectivity. Hence, we choose bigraphs as our spatial model. For example, while the UAV moves from one location to another there is often the need to change control authority from one ground station to another. This manoeuvre is known as handover. A handover manoeuvre is seen as a change of connectivity between an UAV and ground stations.

Symbolic models usually have a small location space compared with physical models. Moreover, locations can be conveniently named turning a spatial model specific to a given domain. Nonetheless, physical information is often needed. For example, to move an UAV from one location to another one needs the GPS location of the destination. Zender et al. [37] and Galindo et al. [38] present hybrid models that combine information at different layers of abstraction, such as geometric, topological, and ontological. Using hybrid models, the programmer has means to model space both symbolically, e.g., a set of names with explicit nested relation, and physically, e.g., augmenting each symbolic name with geometrical coordinates. Note that these hybrid approaches may lead to inconsistencies between the explicit relations modelled symbolically and the implicit relations between the physical interpretation of locations. For example, one can explicitly model a person inside a building while the GPS coordinates of the person are not contained by the GPS-defined polygon that defines the building.

With logical-space computing [39, 40] we take a different approach. Instead of offering the programmer with a hybrid abstraction that blends both physical and symbolical information, programs operate over a symbolic model named logical space, while the runtime system is in charge of interacting with the physical space. The consistency between the logical space
and the physical space is ensured by the logical-space computing semantics. The aim is to keep the simplicity of programming over symbolic spacial abstractions, while still being able to make the desired effects over the physical space.

1.3 Spatial programming models

In this section, we position the BigActor Programming Language in the literature of spatial programming languages. The BigActor Programming Language (BAL) is a spatial programming language that follows the bigActor semantics. It is implemented as an extension of the Scala Actor Library. BAL programs run over a runtime system, named BigActor Runtime System (BARS), which interfaces with an abstract bigraphical machine. BARS is introduced in Chapter 4. BARS logically host a bigActor instance by a bigraph node that models the underlying computing machine. Besides the regular Scala Actor commands (i.e. spawn new bigActors and asynchronously sending messages), a bigActor program can query the bigraphical space model for local observations, request control actions to change the bigraph, and request to migrate from its current host to another host. The interactions with the physical world are formalized by the logical-space computing semantics presented in Chapter 5. The Logical-Space Execution Engine (LSEE), presented in Chapter 6, is a software layer that bridges BARS with a physical world of robots and sensors performing an environmental monitoring mission.

Beal et al. [12] present a survey of spatial computing programming models. The authors survey eight different fields: amorphous computing, biological, agent-based, pervasive computing, robotics, parallel and reconfigurable computing, and formal calculi. The authors compare the models according to their characteristics (e.g. paradigm and target platforms), operators to observe and control the environment, and characteristics of the environment abstraction (e.g. discretization and granularity). Next we review some relevant models and relate them with BAL.

Amorphous computing [41] is concerned with systems of large families of homogeneous, unreliable, locally communicating, simple computation devices. Proto [42, 43] is an amorphous programming language for programming spatial computers using a continuous space abstraction. It is concerned with collections of devices that are distributed to fill the space and where communication capability between devices is strongly coupled with their distance. Rather than specifying the behaviour of individual entities, a proto program specifies the behaviour of regions of space. The programs are automatically distributed to a collection of devices which actions produce an approximation of the aggregate behaviour. BAL does not require computing agents to be homogeneous. Moreover, we are interested in internet-based systems where connectivity between two entities might not be directly related to their physical distance. BAL uses the bigraphs to model space, which provide an abstract model of connectivity. The dynamics is formalized using Bigraph Reaction Rules, which are not necessarily triggered by physical distance.

Borcea et al. [44] introduce the Spatial Programming (SP) language. SP provides high
level abstractions for network-transparent access to data and services distributed across the physical space. The main concept behind SP is the one of spatial references. Spatial references are tuples of the kind \textit{space : name} that model physical geographical regions where computation is embedded. SP runs over a Smart Messages \cite{45} runtime system. Smart messages is a model of concurrency based on migration of computing entities through a shared memory space. Gaia \cite{46} is another relevant spatial computing platform. The Gaia framework provides a runtime environment to support the concept of \textit{active spaces}. An active space is a physical space augmented with computing devices and software to enhance its capabilities. Gaia works as a meta operating system that supports active space applications. Such applications are developed in the context of generic active spaces. Gaia OS adapts the application requirements to the particular properties of its associated physical space, fostering code reusability. With SP and Gaia the programmer has control over both symbolical and physical spaces. With BAL the programmer only has access to a symbolic spaces while the runtime system is in charge of interfacing the physical world. BAL deliberately reduces the expressiveness of the programmer for the sake of a correct physical execution that is ensured by the runtime system.

Biological systems often exhibit locality and spatial structures. MGS \cite{47} is a declarative programming language for simulation of biological processes. Computation in MGS is defined using topological collections that are manipulated using transformation rules. A program in MGS locally manipulates values and modify local structures. Thus, MGS is capable of simulating dynamic systems with dynamical structure. MGS is developed specifically to model biological processes symbolically. BAL programs run over BARS, which interfaces with an abstract bigraphical machine. The bigraphical machine can be physically realized for targeting a variety of systems and applications. In this dissertation we formalized this realization semantically and provide a implementation that targets mobile robotic applications.

One can find in the literature spatial computing languages that emerged from the area of robotics \cite{48}. The amorphous computing language Proto is used to specify the behaviour of swarms of homogeneous robots that are programmed to fill a space \cite{49, 50}. DynaRole \cite{51} and RoCoRo \cite{48} (a generalization of DynaRole inspired by Proto) are languages for modular robotics. Both languages are role-based and provide constructs to specify robot’s local behaviors which are active in given contexts. Konur et al. \cite{52} investigates the use of probabilistic model checking for specifying and formally analyse swarms of robots. The authors specify a foraging robot scenario in PRISM model checker as Discrete-Time Markov Chains (DTMC). The specification is written using the probabilistic temporal logic PCTL. Klavis et al. \cite{53} present a class of graph grammars (also known as graph rewriting systems) to model and control concurrent self-organizing robots. The robotic system is represented as a labelled graph where the nodes of graphs represent robots and labels represent their state while edges represent their interactions. The Collaborative Sensing Language (CLS) \cite{54} is a language for the specification of ad-hoc mobile robot networks. In CSL the programmer specifies the mission (e.g. a sequence of tasks like visiting a location, patrolling a line, survey an area, etc.) in a Petri-net like fashion. The programmer does not specify specific robot behaviours but rather focus on the overall mission specification. The Collaborative Sensing
System takes the mission specification and allocates the available resources to perform the mission according to some notion of optimality (e.g. distance travelled by the robots). The CSL/CSS framework was demonstrated using teams of UAVs performing surveillance tasks. BAL targets heterogeneous robotic systems that requires location and connectivity as first-class citizens. We specifically address the area of unmanned vehicles and present a case study where a programmer uses BAL to specify the spatial behavior of vehicles and sensors performing an environmental monitoring mission. This is presented in Chapter 6.
Part I

Reactive programming on bigraphical machines
Chapter 2

Observing and Controlling Bigraphs

This dissertation addresses computation that is able to observe and to control the space where it is embedded, e.g., a robot observing the environment with its sensors and affecting the environment using its actuators, or a person moving around a city with a smartphone taking pictures.

Our spatial worlds are modelled as bigraphs, including the computing machinery (e.g. robots, smartphones, etc.). Thus, computation addressed in this dissertation, observes and controls bigraphical worlds. For example, BigActors introduced in Chapter 3, are computing entities embedded in spatial dynamic worlds modelled as bigraphs. They can perceive their bigraphical environment by querying for observations and influence the environment by requesting control actions specified as reaction rules.

In this chapter we investigate how to generically observe and control bigraphical worlds. This provides the foundations for modelling the interactions between bigActors and the underlying bigraphical world. The semantic glue that binds bigActors and bigraphs is presented in Chapter 3 where we introduce the BigActor Model and its semantics.

In Section 2.1 we provide a review of the bigraphical model. This review is not intended to be an exhaustive presentation but rather cover the necessary formalism to understand the remainder of this dissertation. For a complete exposition on bigraphs see [3].

In Section 2.2 we introduce a query language to locally observe bigraphs. This approach contrasts with other bigraphical approaches in the literature [36, 19] where both the reactive behaviour and observations are modelled using Bigraph Reactive Systems (BRS).

In Section 2.3 we formalize how to locally control bigraphs.

2.1 Bigraphical formalism - a review

As the name suggests a bigraph is a mathematical structure with two graphs, the place graph - a forest that represents nested locality of components and the linking graph - a hypergraph that models connectivity between components. Figure 2.1 presents an example of a bigraph. The corresponding placing and linking graphs are presented in Figure 2.2.
Figure 2.1: Example of a bigraph.

Figure 2.2: Placing and Linking graphs of the bigraph of Figure 2.1.

Place graphs are contained inside regions\(^1\) and may also contain holes\(^2\). Regions and holes enable composition of placing graphs, i.e. a hole of a given bigraph can be replaced by a region of another bigraph using the composition operator. We explain composition later.

A linking graph may contain hyperedges and inner names and outer names\(^3\). Just as one can fit regions inside holes, one can also merge inner names and outer names using the bigraph composition operator.

A node can have ports\(^4\) which are points for connections to edges or names. The kinds of nodes and their number of ports (arity) are the signature of the bigraph.

The signature takes the form \((K, ar)\) where \(K\) is a set of kinds of nodes called controls and \(ar : K \rightarrow \mathbb{N}\) assigns an arity (i.e. a natural number) to each control. Each node in the bigraph is assigned a control. For example, the bigraph of Figure 2.1 has the following

---

\(^1\)Regions are graphically represented by dashed rectangles and are also known as roots

\(^2\)Holes are graphically represented as gray rectangles and are also known as sites

\(^3\)Names are graphically represented as gray rectangles and are also known as sites

\(^4\)Ports are graphically represented as black dots on the node.
signature: $\mathcal{K} = \{ M : 1, Q : 2 \}$ where $m_i$ has kind $M$ with arity 1 and $q_i$ has kind $Q$ with arity 2. By convention we start kind names with upper-case characters and node names with lower-case characters.

A bigraph $b$ is called concrete when each node and each edge is assigned a unique identifier (known as support). The set of concrete bigraphs with a signature $\mathcal{K}$ is denoted as $\mathcal{Bg}(\mathcal{K})$.

A bigraph without support is called abstract. In abstract bigraphs, nodes are exclusively denoted by their controls. The structure of an abstract bigraph is defined algebraically. The set of abstract bigraphs with a signature $\mathcal{K}$ is denoted as $\mathcal{Bg}(\mathcal{K})$.

We almost exclusively use concrete bigraphs and thus we devote this section to present them formally. For a formal definition of abstract bigraphs see [3], Chapter 3.

**Definition 2.1.** A **concrete bigraph** $b$ is defined as a 5-tuple together with its interface.

$$b = (V, E, \text{ctrl}, \text{prnt}, \text{link})$$

where

- $V$ is a set of node identifiers.
- $E$ is the set of hyperedge identifiers.
- $\text{ctrl} : V \rightarrow \mathcal{K}$ is a control map that assigns controls to nodes.
- $\text{prnt} : m \cup V \rightarrow V \cup n^5$ is the parent map that assigns a parent (a node or a region) to a hole or a node. The set of holes is defined as $m = \{0, \ldots, |m| - 1\}$ while the set of regions is defined as $n = \{0, 1, \ldots, |n| - 1\}$.
- $\text{link} : X \cup P \rightarrow E \cup Y$ is the link map that assigns edges and outer names to inner names and ports. The set of ports $P$ is formalized as $P = \{(v, i) \mid i \in \{0, 1, \ldots, \text{ar} (\text{ctrl}(v)) - 1\}\}$.

Let $\langle m, X \rangle \rightarrow \langle n, Y \rangle$ be the interface of $b$ (denoted as $b : \langle m, X \rangle \rightarrow \langle n, Y \rangle$). $\langle m, X \rangle$ is called the inner face of $b$ and specifies the set of inner names $X$ and the set of holes $m$. $\langle n, Y \rangle$ is called the outer face of $b$ and specifies the set of outer names $Y$ and the set of regions $n$.

**Example 2.1.** Let $b$ be the bigraph depicted in Figure 2.1. $b$ is defined as

$$(V, E, \text{ctrl}, \text{prnt}, \text{link}) : \langle \{0\}, x \rangle \rightarrow \langle \{0\}, y \rangle$$

---

$^5$The symbol $\cup$ denotes the exclusive union operator for sets.
where:

\[ V = \{m_0, m_1, m_2, q_0, q_1\} \]  

\[ E = \{e\} \]  

\[ ctrl(v) = \begin{cases} 
M : 1, & \text{if } v \in \{m_0, m_1, m_2\} \\
Q : 2, & \text{if } v \in \{q_0, q_1\} 
\end{cases} \]  

\[ prnt(v) = \begin{cases} 
0, & \text{if } v = m_2 \\
m_0, & \text{if } v = q_0 \\
m_1, & \text{if } v \in \{q_1, 0\} \\
m_2, & \text{if } v \in \{m_0, m_1\} 
\end{cases} \]  

\[ link(l) = \begin{cases} 
e, & \text{if } l \in \{x, (m_0, 0), (m_1, 0), (q_0, 0), (q_1, 0)\} \\
y, & \text{if } l \in \{(q_0, 1), (q_1, 1), (m_2, 0)\} 
\end{cases} \]  

\[ P = \{(m_0, 0), (m_1, 0), (m_2, 0), (q_0, 0), (q_0, 1), (q_1, 0), (q_1, 1)\} \]  

We use bigraphs to model the location and connectivity of computing machines and their physical environment.

**Example 2.2.** Consider users with smartphones walking on a city. A user can walk from one street to an adjacent one. Streets may contain buildings. Some streets may also contain wireless hotspots which can be connected to the internet. If a smartphone is contained in a wireless hotspot it can connect to it.

In order to model this physical reality using bigraph, first we must define the bigraph signature \( \mathcal{K} \).

\[ \mathcal{K}_{city} = \{\text{Street} : 1, \text{WiFi} : 1, \text{Smart} : 2, \text{Building} : 0\} \]  

The nodes of kind \text{Street} model streets. The connectivity of \text{Street} nodes models their physical adjacency (arity 1). The nodes of kind \text{WiFi} model wireless hotspots. The connectivity of \text{WiFi} nodes models their network topology (arity 1). The nodes of kind \text{Smart} model smartphones. The connectivity of \text{Smart} nodes models its connectivity to wireless hotspots and to 3G network providers (arity 2). The nodes of kind \text{Building} model buildings. \text{Building} nodes have 0 arity and thus do not define a connectivity structure.

Figure 2.3 depicts a bigraph from \( Bg(\mathcal{K}_{city}) \). The figure shows a city with six nodes of kind \text{Street} (grey boxes), namely \text{street0} to \text{street5}. All streets but \text{street5} are connected with a link named \text{road} which models physical adjacency between the corresponding physical locations. Some streets contain nodes of kind \text{WiFi} (blue circle), namely \text{street3} contains \text{wlan0} and \text{street4} contains \text{wlan1}. A smartphone modelled as a node \text{sp} (white star) is located at \text{street2}. A server modelled as a node \text{srv} (green star) is located at \text{street5}. The nodes \text{wlan0}, \text{wlan1}, and \text{srv} all share a link named \text{internet} modelling the connectivity of the corresponding entities to the internet. The nodes \text{street1} and \text{street0}
contains nodes of kind Building, namely build0 to build3. The dark gray box inside the
nodes of kind Building are holes. They model the fact that there might be some structure
inside those nodes (e.g. floors, rooms, corridors, etc.) but it is abstracted away in this
representation.

Note that the signature of the bigraph does not impose any structural constraint rather
than the arity of the nodes. One must be able to specify, for example, that Street nodes
can contain Smart nodes but not the reverse. This kind of structural constraints are defined
using a sorting discipline.

**Sorting**

Milner [3] introduces a typing discipline named *Sorting*. Sorting provides means to con-
strain the kind of bigraphs that generate from a given signature.

**Definition 2.2** (place sorting). A place sorting is defined as $\Sigma = (\Theta, \mathcal{K}, \Phi)$ where $\Theta$ is a
set of sorts, $\mathcal{K}$ is a sorted signature over $\Theta$, and $\Phi$ is the formation rule of $\Sigma$, i.e. a set of
properties that the sorted bigraph place structure must satisfy. By convention, sorts range
over $a, b, \ldots, z$. Disjunctive sorts are denoted as $\hat{ab}$ meaning either sort $a$ or sort $b$.

**Example 2.3.** Let $\Sigma_{\text{city}} = (\Theta, \mathcal{K}_{\text{city}}, \Phi)$ where $\Theta = \{s, w, p, b\}$, $\mathcal{K}_{\text{city}} = \{\text{Street} : s, \text{WiFi} : w, \text{Smart} : p, \text{Building} : b\}$, and $\Phi$ given by:

- an $p$–node is atomic
- all children of a $s$–node have either sort $w$, $p$, or $b$
- all children of a $b$–node have either sort $p$ or $w$
• all children of a \(w\)-node have sort \(p\)
• all children of an \(s\)-root have sort \(s\)
• all children of an \(\hat{pwb}\)-root have either sort \(p\), \(w\), \(b\), or \(\hat{pwb}\)
• all children of an \(\theta\)-root have sort \(\theta\) where \(\theta \in \Theta\)

Operations on bigraphs

Milner introduced two operators for combining bigraphs, the \textit{composition operator} and the \textit{tensor product}.

The composition operator is defined by matching interfaces. Like function composition, a bigraph \(A : I \rightarrow J\) composed with a bigraph \(B : K \rightarrow I\) is a bigraph \(C : K \rightarrow J\).

\textbf{Definition 2.3.} Let

\[ A = (V_A, E_A, ctrl_A, prnt_A, link_A) : \langle m_a, X_a \rangle \rightarrow \langle n_a, Y_a \rangle \]

and

\[ B = (V_B, E_B, ctrl_B, prnt_B, link_B) : \langle m_b, X_b \rangle \rightarrow \langle n_b, Y_b \rangle \]

The \textbf{composition} of bigraphs \(A\) and \(B\), denoted by \(A \circ B\), is a bigraph

\[ C = (V_C, E_C, ctrl_C, prnt_C, link_C) : \langle m_b, X_b \rangle \rightarrow \langle n_a, Y_a \rangle \]

where \(V_C = V_A \uplus V_B\), \(E_C = E_A \uplus E_B\), \(ctrl_C = ctrl_A \uplus ctrl_B\), \(prnt_C\) is obtained by “filling” the holes of \(A\) with regions of \(B\) while \(link_C\) is obtained by “merging” the inner names of \(A\) with the outer names of \(B\). By convention, the regions are matched to holes with the same indices and inner names are matched with outer names with the same name. For a formal definition of \(prnt_C\) and \(link_C\) see [3], page 17. The composition \(A \circ B\) is well defined iff the outer face of \(B\) is equal to the inner face of \(A\), i.e. \(\langle n_b, Y_b \rangle = \langle m_a, X_a \rangle\).

\textbf{Example 2.4.} Consider the bigraphs of Figure 2.4. The bigraph \textit{streetMap} : \(\langle 6, \emptyset \rangle \rightarrow \langle 1, \emptyset \rangle\) models a street map where the grey nodes represent streets and links represent physical adjacency between street nodes. Bigraph \textit{networkInf} : \(\langle 6, \emptyset \rangle \rightarrow \langle 6, \emptyset \rangle\) models a network infrastructure where the blue nodes denote places with network connectivity (local area networks (LAN), wireless LAN, etc.) and links represent connectivity which is linked to name internet denoting the internet. The bigraph resulting from the composition of both bigraphs is \(streetMap \circ networkInf : \langle 6, \emptyset \rangle \rightarrow \langle 1, \emptyset \rangle\). Note that since the \textit{networkInf} kept the holes inside street nodes one could compose \(streetMap \circ networkInf\) with other features (e.g. cars, pedestrians, utilities network, etc.).

Milner also introduced another operator to combine bigraphs called the tensor product.
CHAPTER 2. OBSERVING AND CONTROLLING BIGRAPHS

Definition 2.4. Let $A$ and $B$ be bigraphs that do not share inner or outer names. The tensor product of $A$ and $B$, denoted by $A \otimes B$, is the juxtaposition of both bigraphs where indices of holes and regions of $B$ are made unique with respect to $A$. The names of tensor product are the union of the names of both bigraphs. The tensor product is a bifunctor of a symmetric monoidal category with interfaces as objects and bigraphs as arrows [3].

Dynamics

Consider again the bigraph of Figure 2.3. The bigraph represents a specific snapshot of the world but there is no information on how it can evolve to another bigraph. For example, where can the smartphone $\mathbf{sp}$ move and in what conditions it can connect to the internet. This is specified using Bigraph Reaction Rules (BRR).

A BRR is a tuple $(R, R', \eta)$ where $R$ and $R'$ are bigraphs called respectively redex\(^6\) and reactum. The redex is the portion of the bigraph to be matched and the reactum is the bigraph that replaces the matched portion. $\eta$ is called the instantiation map and indicates how holes in $R$ correspond to holes in $R'$. If $\eta$ is the identity map, then we represent the rule as $R \rightarrow R'$. For the reminder of this thesis we assume $\eta$ to be the identity map, i.e., the

---

\(^6\)Redex stands for “reducible expression”.

---
hole $i$ in $R$ matches with the hole $i$ in $R'$ for all $i \in \mathbb{N}$. If $R$ and $R'$ are abstract bigraphs, then $R \rightarrow R'$ is an abstract BRR. If $R$ and $R'$ are concrete then the BRR is concrete. It is often convenient to define BBRs to be abstract even when working with concrete bigraphs. This defines rules that can be applied on several contexts.

Let $r = R \rightarrow R'$ be a BRR and $B$ a bigraph. In order to perform the reaction $r$ in $B$ we need first to decompose $B$ into $C \circ R \circ d$ where $C$ represents the context and $d$ represents the parameters inside the holes of $R$. Assuming that $\eta$ is the identity map, we compose $C$ with the reactum $R'$ and with $d$ to get the resulting $B'$, i.e., $B = C \circ R \circ d \Rightarrow B' = C \circ R' \circ d$.

The application of BRRs works both for abstract and concrete BRRs. If a bigraph is concrete then the redex and the reactum of the BRR must also be concrete. We use abstract BRR when we want to evidence the generality of a given rule. Nonetheless, since we work with concrete bigraphs, the rule must be concretized, i.e., one must provide the redex and reactum with a support before the application.

**Example 2.5.** Consider the two abstract BRR of Figure 2.5. The first rule, MOVE, models a computational device, e.g., a user with a smartphone denoted by a star with kind Smart, moving from one street to another. The second rule, CONNECT, models computational device to move inside a hotspot with network connectivity denoted by a node with kind Wlan) and connecting to the network. Note that the rules are parametric, i.e., they can be applied regardless the nodes inside the street nodes. For example, this rule allows moving a node of kind Smart regardless of the content of the node of kind Street. Figure 2.6 shows an example of the application of BRR MOVE. The bigraphs in this example are concrete. Thus, we need to first concretize MOVE. Since we want to move sp0 from its current location to street3 we denote the concrete BRR as MOVE(sp0, street3). Figure 2.6 depicts the context $C$, and parameters $d$ where the rule is applied. Note that the parameters allow the rule to be applied with sp1 inside street3. At the top of Figure 2.6 shows the bigraph $B$ transitioning to $B'$. At the bottom we show the decomposition of each bigraph in the context $C$, redex $R$, reactum $R'$, and parameters $d$. 
Definition 2.5. A bigraph reactive system (BRS), denoted as $\text{Bg}(\mathcal{K}, \mathcal{R})$, is a bigraphical category $\text{Bg}(\mathcal{K})$ and a set of BRR $\mathcal{R}$.

Next we define a trace of a BRS based on [55].

Definition 2.6. A trace of a BRS $(B, \mathcal{R})$ is a sequence of bigraphs

$$t = (B_0, B_1, \ldots)$$

where $B_0 = B$ and for each $B_i$ and $B_{i+1}$ there exists a reaction rule $R \rightarrow R' \in \mathcal{R}$ a context $C$ and parameters $d$ such that for each $i$

$$C_i \circ R \circ d_i \rightarrow C_i \circ R' \circ d_i$$

where $B_i = C_i \circ R \circ d_i$ and $B_{i+1} = C_i \circ R' \circ d_i$. The set of all traces starting from $B$ and generated by rules $\mathcal{R}$ is denoted by $\mathcal{T}_B^R$.

Example 2.6. Consider the bigraph depicted in Figure 2.3. The application of the reaction rules $\text{MOVE}(\text{sp}, \text{street1})$, $\text{MOVE}(\text{sp}, \text{street3})$, and $\text{CONNECT}(\text{sp}, \text{wlan0})$ results in the execution trace presented in Figure 2.7.
CHAPTER 2. OBSERVING AND CONTROLLING BIGRAPHS

2.2 Querying for local observations

In this section we introduce a query language for bigraphs. Queries provide means for agents to specify local observations of the bigraph with respect to their location. An agent requesting to observe the bigraph is denoted as `host`. One can think of a host node as denoting the physical entity that is querying the bigraphical world, i.e., hosting the computation that is querying the world. We think of “local” in terms of the placing graph (parents and children of a given node) and in terms of the linking graph (nodes linked to a given node).

The query language syntax is specified by the following grammar:

\[
\text{query ::= node | CHILDREN(node) | LINKED_TO(node)}
\]

\[
\text{node ::= HOST | PARENT(node)}
\]

For example, the query `CHILDREN(PARENT(HOST))` can be interpreted in English as “the children of the parent of the host”.

Queries are interpreted as bigraphs. The interpretation of a query \( q \in \text{query} \) over a bigraph \( B \) with respect to a host \( h \) is formalized by the semantic function

\[
\llbracket \cdot \rrbracket^B_h : \text{query} \rightarrow \mathcal{B}g(\Sigma).
\]
Given a query \( q \in \text{query} \), the interpretation \( [q]^B \) is composable with a context in \( B \), i.e., there exists a context \( C \) and parameters \( d \) such that \( B = C \circ [q]^B \circ d \).

**Example 2.7.** Figure 2.8 presents an example of querying the bigraph \( B_0 \) of Figure 4.7 with respect to \( sp \). On the right-hand side of Figure 2.8 one can see the interpretation of the queries \( \text{HOST}, \text{PARENT(\text{HOST})}, \text{CHILDREN(\text{PARENT(\text{HOST})})} \).

![Figure 2.8: BigActor observing the street map using query \text{CHILDREN(\text{PARENT(\text{HOST})})}](image)

The semantic function \( [\cdot]^B \) is defined inductively over the structure of the query grammar.

Every query is defined with respect to a node named \( \text{HOST} \). Thus \( [\text{HOST}]^B \) defines the base case.

\[
[\text{HOST}]^B \mapsto ([\text{\{h\}}, \emptyset, \text{ctrl}_h, \text{prnt}_h, \text{link}_h] : (1, \emptyset) \rightarrow (1, Y_h)
\]
where:

\[
\begin{align*}
\text{ctrl}_h(h) &= \text{ctrl}_B(h) \\
\text{prnt}_h(v) &= \begin{cases} 0 & \text{if } v = h \\ h & \text{if } v = -0 \end{cases} \\
\text{link}_h(l) &= \begin{cases} \text{link}_B(l) & \text{if } l \in Y_B \\ y_e & \text{if } e = \text{link}_B(l) \in E_B \end{cases}
\end{align*}
\]

\[
Y_h = \{ y_e | \forall e = \text{link}_h(l) \land e \in E_B \} \cup \{ y | \forall l. y = \text{link}_h(l) \land y \in Y_B \}
\]

\([\text{HOST}]_h^B\) is a bigraph with a single node (i.e. \(h\)) with the same arity as in \(B\). The node \(h\) contains hole \(-0\). For each edge \(e\) connected to \(h\) in \(B\) there is a outer name \(y_e\) in \([\text{HOST}]_h^B\) connected to \(h\). These outer names are introduced such that the composition \(B = C \circ [\text{HOST}]_h^B \circ d\) is valid. The remaining outer names of \(B\) connected to \(h\) are kept in \([\text{HOST}]_h^B\).

The first inductive definition is \([\text{PARENT}(\text{node})]_h^B\). Let \([\text{node}]_h^B = B_n\) where \(B_n\) is a bigraph with a single node \(n\).

\[
[\text{PARENT}(\text{node})]_h^B \mapsto \{ p, \emptyset, \text{ctrl}_p, \text{prnt}_p, \text{link}_p \} : \langle 1, \emptyset \rangle \rightarrow \langle 1, Y_n \rangle
\]

where

\[
\begin{align*}
p &= \text{prnt}_B(n) \\
\text{ctrl}_p(p) &= \text{ctrl}_B(p) \\
\text{prnt}_p(v) &= \begin{cases} 0 & \text{if } v = p \\ p & \text{if } v = -0 \end{cases} \\
\text{link}_h(l) &= \begin{cases} \text{link}_B(l) & \text{if } l \in Y_B \\ y_e & \text{if } e = \text{link}_B(l) \in E_B \end{cases}
\end{align*}
\]

\[
Y_n = \{ y_e | \forall e = \text{link}_h(l) \land e \in E_B \} \cup \{ y | \forall l. y = \text{link}_h(l) \land y \in Y_B \}
\]

\([\text{PARENT}(\text{node})]_h^B\) is a bigraph with a single node, the parent of \(h\). The linking structure is defined in the same way as in \([\text{HOST}]_h^B\).

The next inductive definition is \([\text{CHILDREN}(\text{node})]_h^B\).

\[
[\text{CHILDREN}(\text{node})]_h^B \mapsto (V_C, \emptyset, \text{ctrl}_C, \text{prnt}_C, \text{link}_C) : \langle |V_C|, \emptyset \rangle \rightarrow \langle 1, Y_C \rangle
\]

where

\[
\begin{align*}
V_C &= \{ v | v \in V_B, n = \text{prnt}_B(v) \} \\
\text{prnt}_C(v) &= \begin{cases} 0 & \text{if } v \in V_C \\ v' & \text{if } v = \text{holeOf}(v') \end{cases}
\end{align*}
\]
holeOf : \( V_C \rightarrow \mathbb{N} \) is a function that assigns to each children node a unique hole to abstract its contents. \( ctrl_C, \ link_C \), and the set of external names \( Y_C \) are given as before although now ranging over the set of nodes \( V_C \) and respective ports. At last, \([\mathsf{LINKED\ TO}(\text{node})]\)^{B}_h\) is inductively defined as follows.

\[
[\mathsf{LINKED\ TO}(\text{node})]\)^{B}_h \mapsto (V_L, \emptyset, ctrl_L, prnt_L, link_L) : \langle |V_L|, \emptyset \rangle \rightarrow \langle |V_L|, Y \rangle
\]

where,

\[
V_L = \{v \in V_B \mid \exists p \in \text{ports}(v). \exists p' \in \text{ports}(n). \text{link}_B(p) = \text{link}_B(p')\}
\]

\[
prnt_h(v) = \begin{cases} \text{regionOf}(v) & \text{if } v \in V_L \\ v' & \text{if } v = \text{holeOf}(v') \end{cases}
\]

The set of nodes \( V_L \) is generated by searching the nodes of \( V_B \) that are connected to \( n \), i.e., the nodes in \( V_L \) have a port that is linked to a port of \( n \). For each node in \( V_L \) we create a unique region using the function \( \text{regionOf} : V_L \rightarrow \mathbb{N} \). By putting each node in a unique region we can restore the parenthood by composing again with a context of \( B \). The remainder of the bigraph definition is as per the previous cases.

**Definition 2.7** (Local match). We say that a bigraph \( B' \) is local match of \( B \) with respect to \( h \) if there exists a query \( q \) such that \( B' = [q]_h^B \).

**Example 2.8.** Consider the host to be \( \text{srv} \). The interpretation of \([\mathsf{LINKED\ TO}(\text{HOST})]\)^{B}_\text{srv}\) is given by:

\[
(V_L, \emptyset, ctrl_L, prnt_L, link_L) : \langle 3, \emptyset \rangle \rightarrow \langle 3, \{\text{y\_network}\} \rangle
\]

where \( V_L = \{\text{wlan0}, \text{wlan1}, \text{wlan2}\} \), \( \forall v \in V_L, ctrl_L(v) = \text{Wlan}, \forall i \in \{0, 1, 2\}, prnt_L(i) = \text{wlan}, \forall i \in \{0, 1, 2\}, prnt_L(\text{wlan}) = i, \forall v \in V_L, link_L((v, 0)) = \text{y\_network} \).

The query language only requires local interpretation with respect to the host node, i.e., the interpretation of a given query does not require the knowledge of the overall bigraph. In other words, the time complexity of querying depends on the size of the query and not on the size of the bigraph.

**Theorem 2.1** (Complexity of querying). The time complexity of interpreting a query into an observation depends on the size of the query (the number of terminal symbols in the abstract syntax tree) and not on the size of the bigraph (the total number of nodes and edges).

**Proof.** In order to calculate the complexity of interpreting a query we need to define a data structure for Bigraphs. We assume that the placing graph is stored in a data structure composed by a list of trees \( P = [T_0, T_1, \ldots, T_n] \) where \( T_i = (\text{root}_i, \text{forest}_i) \) where \( \text{root}_i \) is a node name and \( \text{forest}_i \) is zero or more child trees. We assume that all subtrees are stored
in a hash table where the key is the root and the value is the corresponding children. The linking graph is stored in a hypergraph \( L = (V, \text{hyper}) \) where \( V \) is the list of vertices and \( \text{hyper} \) is a list of lists of vertices representing the hyperedges. Vertices are also annotated with their arity.

By structural induction of \( \text{query} \).

- Base case (\( \text{HOST} \)): Interpreting \( \text{HOST} \) requires reading a variable. This takes \( O(1) \) and thus does not depend on the size of the bigraph.

- Inductive steps: The time complexity of interpreting \( \text{PARENT}(\text{node}) \) is equal to the complexity of interpreting \( \text{node} \) plus the complexity of getting the root node which is equal to \( O(1) \). Since \( \text{node} \) is either \( \text{HOST} \) or recursive application of \( \text{PARENT} \) the complexity depends linearly on the size of the query (\( O(n) \) where \( n \) is the size of the query) and not on the size of the bigraph. The time complexity of interpreting \( \text{CHILDREN}(\text{node}) \) is equal to the time complexity of interpreting \( \text{node} \) (\( O(n) \)) plus the complexity of getting the children nodes. Since all subtrees are saved in a hash table, getting the children is of complexity \( O(1) \). The time complexity of interpreting \( \text{LINKED}_\text{T0}(\text{node}) \) is equal to the time complexity of interpreting \( \text{node} \) plus the complexity of looking for hyperedges containing the node. This depends on the arity of the node (which is constant and stated initially in the bigraph signature).

\[
T(\text{query}) = n \times \text{max(arity)} = O(n) \text{ where } n \text{ is the size of the query.}
\]

The complexity of the interpretation of a query is on the size of the query and not on the size of the overall bigraph comes from the fact that the interpretation is local with respect to a host node.

**Corollary 2.1.** The time complexity of interpreting a query into an observation depends linearly on the size of the query.

**Proof.** This theorem comes as a corollary of Theorem 2.1. By structural induction and assuming that interpreting the base case \( \text{host} \) takes constant time, then interpreting \( \text{PARENT}(\text{query}) \) or \( \text{CHILDREN}(\text{query}) \) takes on extra step (looking up one level in the place tree or looking on level down), while interpreting \( \text{LINKED}_\text{T0}(\text{query}) \) takes in the worst-case the maximum arity declared in the bigraph signature (which is a constant value). Thus, the worst-case complexity \( T(\text{query}) = n \times \text{max(arity)} = O(n) \) where \( n \) is the size of the query.

### 2.3 Local bigraphical reaction rules

We assume that agents affect their environment in the vicinities of their hosts. In bigraphical worlds, we enforce locality by imposing constraints on the reaction rules requested by an agent with respect to the location of its host.
We use the query language to define which part of the bigraph a given BRR can change. Since the query language defines local observations with respect to a host node, one can extend the concept of locality to reaction rules. The verification that a BRR $R \rightarrow R'$ satisfies a locality property specified by a query $q \in \text{query}$ is performed by the following function:

$$
local_q : \text{Bg}(\Sigma) \times \text{Bg}(\Sigma) \rightarrow \mathbb{B}
$$

$$
local_q(R, R') \mapsto \begin{cases} 
\text{true} & \exists S. R = S \circ \hat{R} \land R' = S \circ \hat{R}' \\
\text{false} & \text{o.w.}
\end{cases}
$$

where $\hat{R} \rightarrow \hat{R}'$ is a reaction rule such that $\hat{R} = \{ q \}_h^B$. The function $local_q$ requires that a reaction rule $R \rightarrow R'$ can only change $\hat{R} = \{ q \}_h^B$ keeping the remaining context $S$ static.

**Example 2.9.** Consider a controller hosted at an autonomous vehicle. The controller is only allowed to move its own host from one location to another keeping the remaining environment untouched. This can be formalized by stating that the BRRs that the controller can use to control the world can only change the world matched by $\{ \text{HOST} \}_h^B$, where $B$ is the current bigraph and $h$ is the node denoting the autonomous vehicle.

**Example 2.10.** Now consider an autonomous vehicle that has the ability move around and manipulate the world around it (e.g. collect tokens). This can be formalized by stating that a controller hosted at the autonomous vehicle can only change the world matched by $\{ \text{HOST} \}_h^B$ or $\{ \text{CHILDREN} \text{(PARENT(HOST))} \}_h^B$, meaning that it can change the host location or the siblings of the host.

### 2.4 Final remarks

In this chapter we provide a review of the bigraphical model and introduce a query language for performing local observations over bigraphs.

The bigraphical formalism was introduced by Robin Milner as the ubiquitous abstract machine, the equivalent to the von Neumann machine for ubiquitous systems [3]. The model entails two graphical structures to model respectively the location and connectivity of ubiquitous systems. The bigraphical formalism provides means for changing one bigraph to another. This is known as Bigraph Reaction Rules (BRRs). BRRs can be seen as the specification of control actions performed over bigraphs forming a bigraphical execution trace.

As Birkedal et al. claimed [36], performing observations over bigraphs under solely the realm of the bigraphical formalism is not practical since one would have to use BRRs which, by definition, change the bigraph being observed. To overcome this difficulty we introduce a query language for performing observations over bigraphs. The query language provides means for querying the placing graph, the linking graph, or both. We demonstrate that the
complexity of interpreting queries from our query language depends on the size of the query and not on the size of the underlying bigraph.

We are interested in modelling mobile computing machines that only affect the environment in their vicinities, i.e., an agent can affect the world locally with respect to its host. We use the query language to define locality and enforce BRRs to only change the bigraph locally with respect to a given host.
Chapter 3

BigActors

Computing agents are getting ubiquitously embedded in spatial environments. They move, interact within each-other, and interact with their environment by observing it and controlling it. These computing agents are carried by machines such as robots, autonomous vehicles, or even people walking with their smartphones. Computation performed at these kind of systems is fundamentally coupled with their environment. In this chapter we introduce the BigActor Model - a model for structure-aware computation. BigActors are computing agents modelled as actors that are embedded in an spatial environment modelled as bigraphs.

This chapter is dedicated to the specification of the bigActor semantics. In Section 3.1 we provide a brief introduction to the BigActor model using a simple example that gives emphasis to its first-class concepts.

The semantics of the BigActor model is inspired by the Actor model operational semantics. In Section 3.2, we provide a brief review of the Actor model and its formal semantics. In Section 3.3 we introduce the BigActor model and its formal semantics. Section 3.6 discusses the correctness of the BigActor Model semantics and presents sufficient conditions for safety.

3.1 Introduction

We introduce the BigActor model in [1] as a model for structure-aware computation. BigActors are Actors as per Hewitt and Agaha’s semantics [2] that operate over Milner’s bigraphical [3] abstractions of the world. A bigActor is an actor hosted by a bigraph node denoting the computing machine that is executing it. Like a regular actor, a bigActor can create new bigActors and send messages asynchronously to other bigActors. In addition it can request observations the bigraph, request control actions to change the bigraph, and migrate from one host to another.

Example 3.1. Figure 3.1 depicts a set of bigActors interacting with a bigraphical model of the world. The picture provides examples for the interactions between bigActors and their bigraphical environment, i.e. send a message to another bigActor, create a new
bigActor, observe the underlying bigraph, request a control action to change the bigraph, and request to migrate to another host. There are three bigraph nodes modelling robots, namely robot0, robot1, and robot2. The robot modelled by node robot0 is located at room0 of building0. The robot modelled by node robot1 is located at room1 of the same building. The robot modelled by node robot2 is located at building1. The link named wifi models the wifi connectivity between robots.

The bigActor system is composed by three bigActors, a0, a1, and a2, that are specified in Figure 3.2. The code is written in the Scala BigActor Programming Language which is explained in detail in Chapter 4.

The bigActor a0 is hosted by robot0 and its objective is to gather all the robots at the location of its host. Its first step is to observe where its host is located. The bigActor a0 requests the observation using the command \texttt{observe(PARENT\_HOST)} which queries the bigraph for the parent of the host of a0 (which in this case is room0). The observation is asynchronous. When the result of the observation is available, it is assigned to the variable \texttt{obs} inside the react body\footnote{The \texttt{react} command is inherited from the Scala Actors library and is originally used for asynchronous message-passing.}. The second step of a0 is to send a message to a1 with the former observation. The message received by a1 gets assigns to the variable \texttt{msg} inside the react body. Later, a1 requests its host to move the location specified in the message. This is
val a0 = BigActor hosted_at robot0 with_behavior{
    observe(PARENT(HOST))
    react{
        case obs => {
            a1 ! obs
            val a2 = BigActor hosted_at robot2 with_behavior{
                control(MOVE_HOST_TO(obs))
            }
        }
    }
}

val a1 = BigActor hosted_at robot1 with_behavior{
    react{
        case msg => control(MOVE_HOST_TO(obs))
    }
}

Figure 3.2: Specification of the bigActor system for Example 3.1.

preformed by a command control(MOVE_HOST_TO(msg)) that applies a Bigraph Reaction Rule over the current bigraph in order to move the host of the bigActor to the location specified by msg. Assume that a0 does not know the name of any bigActor hosted at robot2. To overcome this situation, a0 synthesizes a new bigActor named a2 that migrates to robot2. The specification of a2 has only one command that requests to move its host to room0.

Figure 3.3 depicts the execution trace for the Example 3.1.

Example 3.1 presents informally the execution semantics of a simple bigActor system. The remainder of the chapter is concerned with providing a formal semantics for the model and analyse its correctness.

### 3.2 Actor model of computation - review

The semantics of the BigActor model is formalized by extending the Actor model operational semantics presented by Agha in [2, 56]. In this section we provide an overview of the Actor model and its semantics with sufficient detail in order to understand the BigActor model semantics. For an exhaustive presentation of the actor model and its semantics refer to [2, 56].

The Actor model of computation is a model of concurrency for distributed concurrent computing entities. An actor system is composed of autonomous objects called actors. Actors have local memory and local behavior. Actors communicate using asynchronous message
passing. Messages that have been sent but not yet received are queued up in the receiver actor’s mailbox. The receiver eventually removes the message and processes it and affecting its behavior. An actor encapsulates a state and a thread. Each actor has a mail-address used by other actors for communication.

An actor reacts upon receiving a message from another actor by: **compute** and change state; **send** a message to another actor; or **create** a new actor. Figure 3.4 depicts an example of an actor system. Figure 3.4 contains three actors, a₀, a₁, and a₂. Actor a₀ starts at state x₀. When it receives a message m it sends the message to a₁ and moves its state to x₁. When a₀ receives the message m’ it creates a new actor a₂.

The Actor model has been adopted as the concurrency model in several modern programming languages such as Erlang [57], Scala [58], Dart, Cloud Haskell [59], and in embedded systems design using Ptolemy II [60]. The use of asynchronous message-passing contrasts with other concurrency models such as **threads**, where computing entities communicate using shared state.
Next we present an operational semantics for the actor model. We follow the operational semantics style taken in [61, 56, 62].

**Operational Semantics**

One can find in the literature several approaches for defining the actor model semantics. Agha provides in [2] a denotational semantics for actors. The same author provides a contextual operational semantics in [56]. The latter is then adapted in [61] to include real-time constraints.

The Actor model operational semantics is formalized as a transition relation over the set of actor configurations.

**Definition 3.1 (Actor Configuration).** An actor configuration is a tuple $\langle \alpha \mid \mu \rangle$ where $\alpha$ is a set of actors, and $\mu$ is the set of pending messages. $\text{Dom}(\alpha)$ denotes the set of unique actor names that identifies each actor in $\alpha$.

An actor configuration is syntactically defined according to the grammar presented in Figure 3.5. The syntax presented at Figure 3.5 is not enforced by the bigActor model. Its only purpose is to introduce syntactic notation in order to introduce the semantics of the model.

The notation $[E \vdash b]_a$ denotes an actor with unique name $a \in \text{Dom}(\alpha)$, local environment $E \in \mathcal{E}$, and local behaviour $b$. The local behaviour is specified as a sequential composition of actor commands, namely, $\text{send}$, $\text{ready}$, and $\text{new}$, and expressions $e_\lambda$ that manipulate the
config : \langle actor^* | msg^* \rangle
actor : \quad \lbrack E \vdash b \rbrack_a
\quad b : \begin{array}{ll}
\text{send}(a, v) \mid \text{ready}(x) \mid \text{new}(b) \mid b \mid e_{\lambda} \mid \text{nil}
\end{array}
msg : \langle a \leftarrow v \rangle

Figure 3.5: Grammar for actors syntax.

state of the actor. The behavior can also take the value nil. The notation \langle a \leftarrow v \rangle denotes a message to an actor \(a \in A\) with content \(v \in V\).

The semantics of the actor model is presented in Figure 3.6 and is an adaptation of the semantics introduced in [61, 56, 62]. The actor model is a model of concurrency. The

\begin{align*}
\langle \text{nil} \rangle & \quad \langle \alpha, [E \vdash \text{nil}]_a \mid \mu \rangle \rightarrow \langle \alpha \mid \mu \rangle \\
\langle \text{fun} : a \rangle & \quad \frac{[E \vdash e_{\lambda}; b]_a \rightarrow_{\lambda} [E' \vdash b]_a}{\langle \alpha, [E \vdash e_{\lambda}; b]_a \mid \mu \rangle \rightarrow \langle \alpha, [E' \vdash b]_a \mid \mu \rangle}
\langle \text{new} : a, a' \rangle & \quad \langle \alpha, [E \vdash \text{new}(b') \vdash b]_a \mid \mu \rangle \rightarrow \langle \alpha, [E \vdash b]_a, [E \vdash b'_{\lambda}]_a \mid \mu \rangle
\langle \text{snd} : a, (a' \leftarrow v) \rangle & \quad \langle \alpha, [E \vdash \text{send}(a', v) \vdash b]_a \mid \mu \rangle \rightarrow \langle \alpha, [E \vdash b]_a \mid \mu, (a' \leftarrow v) \rangle
\langle \text{rcv} : a, (a \leftarrow v) \rangle & \quad \langle \alpha, [E \vdash \text{ready}(x) \vdash b]_a \mid \mu, (a \leftarrow v) \rangle \rightarrow \langle \alpha, [E[x \mapsto v] \vdash b]_a \mid \mu \rangle
\end{align*}

Figure 3.6: Actor operational semantics.

semantics of internal computation of actors is not addressed explicitly by the model and, as such, it is left unspecified. For the sake of a complete formalization the actor model semantics assumes that internal actor computation is modelled by the transition relation \(\rightarrow_{\lambda}\) that is given by the semantics of an arbitrary language that is used for specifying the internal behaviors of actors. In order to make the analysis of the actor semantics more readable we assume that the language that specifies the actors internal behavior has a sequential composition operator where denoted by \(\rightarrow\). This approach is used by Nielson et al. [61]. For an actor semantics without the need for a sequential composition see [56].

The semantics consists of five inference rules. The rule \langle \text{nil} \rangle models the behaviour of an actor \(a\) with behaviour \text{nil}. The rule simply removes \(a\) from the set of actors. The rule \langle \text{fun} : a \rangle models an internal computation of an actor \(a\). It takes an expression \(e_{\lambda}; b\), executes \(e_{\lambda}\), produces the corresponding side-effects in the local state, and changes the behaviour of
CHAPTER 3. BIGACTORS

<table>
<thead>
<tr>
<th>Actor commands</th>
<th>Scala Actors</th>
</tr>
</thead>
<tbody>
<tr>
<td>send(a,m)</td>
<td>a ! m</td>
</tr>
<tr>
<td>ready(x)</td>
<td>react{</td>
</tr>
<tr>
<td></td>
<td>case x =&gt; ...</td>
</tr>
<tr>
<td>new(b)</td>
<td>actor{b}</td>
</tr>
</tbody>
</table>

Table 3.1: Encoding of Scala actor commands into the actors grammar.

In this thesis we use Scala Actors [23] as the implementation of the actor model. Scala uses actors as its de facto concurrency model. Table 3.1 shows the syntactic encoding of the actor commands `send`, `ready`, and `new` in the Scala Actor commands “!”, `react`, and `actor`.

Example 3.2. Consider a simple social network example. There are two actors, `app` and `social`. `app` sends messages of kind `(user, status)` to `social`. The message is a tuple where the first element is the name of the user and the second is its status. `social` local state contains a map named `statusMap` which contains pairs of users and their corresponding status. When `social` receives a message of kind `(user, status)` it updates the value `user` with the new status. Otherwise, it creates a new key-value pair in the map. Figure 3.7 presents the Scala code for `app` and `social`.

The encoding of `app` and `social` into the actor syntax is respectively

\[ [\{\} \vdash \text{send}(\text{social},("Nina","Feeling good");\text{nil})]_{\text{app}} \]
\[ [\text{statusMap} \vdash \text{ready}(x);e_\lambda;\text{nil}]_{\text{social}} \]

The execution trace of the actor system \{`app`, `social`\} is given in Figure 3.8.

The variable \(x\) is locally scoped under the context of the `react` body and can be pattern-matched using case alternatives.
CHAPTER 3. BIGACTORS

```scala
val app = actor{
  social ! ("Nina","Feeling good")
}

val social = actor{
  val statusMap = Map("Nina" -> "Put a spell on you",
                       "Miles" -> "Kind of Blue")
  react{
    case (user,status) => statusMap(user) = status
  }
}
```

Figure 3.7: `app` and `social` actors.

### 3.3 BigActor Model

Recall Example 3.2. The actor system formed by `app` and `social` specifies concurrent computation. The actor `app` sends asynchronously a message to `social` which reacts upon the reception of such message. In this actor system there is no explicit information of where computation is being executed. For example, `app` might be hosted at a smartphone that only has connectivity in certain parts of the city. One would benefit from an `app` actor that only sends updates to `social` when it has network connectivity. Or that would send directions to the smartphone user to move toward an area of the city where it is known to have a wireless hotspot where it could connect. In a pure actor system, these interactions between `app`, its host (the smartphone), and its environment are not explicit. They would have to be modelled as actors. For example, the smartphone could be modelled as an actor that keeps the location and connectivity information and interacts with another actor that models the spatial environment of the city. The choice of the spatial model and its correct implementation is in the hands of the programmer since the actor model lacks any explicit spatial abstractions.

**Example 3.3.** Figure 3.9 shows the Scala code for a new actor system for the scenario presented in Example 3.2. The new code for `app` sends a message to `smartphone` asking if there is network connectivity. Next `app` waits for the reply. If the answer is positive then `app` proceeds by sending the status update to `social`. Otherwise, `app` requests `smartphone` to navigate to another location (for the sake of simplicity we assume that there are only two locations, `loc0` and `loc1`). The actor `smartphone` interacts with the actor `env` modelling the environment. `smartphone` keeps a record in its local state about its current connectivity and location. It replies to requests from `app` asking for the connectivity state and interacts with `env` that models the spatial environment and connectivity. The actor `env` keeps track of the smartphone location and connectivity and models their dynamics. The connectivity is check using an API call to a wifi driver. The code for `social` is the same as per Example 3.2.
Figure 3.8: Execution trace of the actor system \( \{ \text{app}, \text{social} \} \).
val app = actor{
    loop{
        smartphone ! "connected?"
        react{
            case true => social ! ("Nina","Feeling good")
            case false => smartphone ! "navigate"
        }
    }
}

val smartphone = actor{
    var location = loc0
    var connected = false
    loop{
        react{
            case "connected?" => app ! connected
            case "navigate" => env ! "move"
            react{
                case (newLoc,newConn) =>
                    location = newLoc
                    connected = newConn
            }
        }
    }
}

val env = actor{
    smartphoneLoc = loc0
    smartphoneConn = false
    loop{
        react{
            case "move" => {
                smartphoneLoc = loc1
                smartphoneConn = wifi.isConnected
                smartphone ! (smartphoneLoc,smartphoneConn)
            }
        }
    }
}

Figure 3.9: Actor system for Example 3.2 with new actors app, smartphone, and env.

With the bigActor model we make the interactions between actors and their spatial environment explicit and formal. A bigActor system runs over a bigraphical model of space.
CHAPTER 3. BIGACTORS

The spatial model abstracts not only the location of components but also their connectivity. This follows the Milner's vision of space and mobility for ubiquitous systems [3].

**Example 3.4.** Figure 3.9 shows the Scala code for a bigActor system for the scenario discussed in Example 3.3.

```scala
val app = BigActor hosted_at smartphone with_behavior{
    loop{
        observe(LINKED_TO(HOST))
        react{
            case obs.contains(srv) => social ! ("Nina","Feeling good")
            case _ => {
                control(MOVE_HOST_TO(street3))
                control(CONNECT_HOST_TO(wlan0))
            }
        }
    }
}

val social = BigActor hosted_at srv with_behavior{
    val statusMap = Map("Nina" -> "Put a spell on you",
                        "Miles" -> "Kind of Blue")
    react{
        case (user,status) => statusMap(user) = status
    }
}
```

Figure 3.10: BigActor system for scenario presented in Example 3.3.

The bigActors run over an environment modelled as the bigraph of Figure 2.3.

Note that the bigActor system of Figure 3.10 has two bigActors, **app** and **social**, just as the original actor system of Figure 3.7). There is no need of extra actors to model the environment and the machines as in the actor system of Figure 3.9.

The bigActor constructor has a new parameter: **hosted at loc**. This parameter defines where the bigActor instance is going to be initially hosted. The bigActor **app** is hosted at the bigraph node **smartphone** denoting the smartphone, while **social** is hosted at **srv** denoting the server where the social network is running.

Comparing with its actor counterpart provide by Figure 3.7, the bigActor **app** performs two new kind of commands. It first observes the structure of the world using the query **LINKED_TO(HOST)**, which queries the bigraph for the nodes connected to the host node, i.e., **app**. The query language used by BigActors to query bigraphs is described in detail in Chapter 2. Upon receiving the observation, **app** checks if **srv** is contained in the observation. If it is, then it sends the message with the updated status. Otherwise it requests to change location and connect to a wireless local area network using, respectively, the commands
control(MOVE_HOST_TO(street3)) and control(CONNECT_HOST_TO(wlan0)). The control commands are Bigraph Reaction Rules (BRR). The application of BRRs over bigraphs is explained in detail in Chapter 2. The BRRs MOVE_HOST_TO(street3) and CONNECT_HOST_TO(wlan0) result from the instantiation of the BRRs presented in Figure 3.11. The instantiation replaces HOST by app, Loc by street3, and Wlan by wlan0.

The commands observe(LINKED_TO(HOST)), control(MOVE_HOST_TO(street3)), and control(CONNECT_HOST_TO(wlan0)), specify formally and explicitly the interactions between the bigActor app and the underlying bigraph. They model, respectively, an observation using a query language and two control actions using bigraph reaction rules. The correct manipulation of space is no longer in the hands of the programmer but rather specified in the semantics of bigraphs and bigraph reaction rules.

Note that both observations and control commands are specified without any reference to smartphone but with respect to the host of the bigActor. Thus, the same bigActor code can be reused with other hosts, e.g., one could create a new app hosted by a tablet by simply replacing smartphone by tablet in the code at Figure 3.10.

Note that the actors smartphone and env of Figure 3.9 contain spatial information in their local state, e.g., the location and connectivity of the smartphone. The programmer must ensure that this information is consistent at all time. This is not needed in the bigActor example since all the spatial information is entailed in the bigraph and can be queried by different bigActors at different instances of time.

Next we introduce the formal semantics for the BigActor model.

### 3.4 BigActors Operational semantics

We define the semantics of the BigActor model as an extension of the Actor model operational semantics defined in Section 3.2. A bigActor configuration extends an actor
Definition 3.2 (BigActor Configuration). A BigActor configuration is a tuple \( \langle \alpha \mid \text{host} \mid \mu \mid \eta \mid B \rangle \) where \( \alpha \) is a set of bigActors, \( \text{host} : \text{Dom}(\alpha) \to V_B \) is the hosting function, \( \mu \) is a set of pending messages, \( \eta \) is a set of pending requests, and \( B \) is a bigraph.

The set \( \alpha \) is a set of bigActors, just as in the actor model it is a set of actors. The set of pending messages \( \mu \) is exactly as in the actor model. The set of pending request \( \eta \), the hosting map \( \text{host} \) and the bigraph \( B \) are specific to the bigActors semantics.

A bigActor configuration is syntactically defined according to the grammar presented in Figure 3.12.

\[
\begin{align*}
\text{config} : & \quad \langle \text{bigActor}^* \mid \text{h}^* \mid \text{msg}^* \mid \text{req}^* \mid B \rangle \\
\text{bigActor} : & \quad [E \vdash b]_a \\
\text{b} : & \quad \text{cmd} | \text{ready}(x) | \text{new}(b) | b; b | e_{\lambda} | \text{nil} \\
\text{cmd} : & \quad \text{send}(a, v) | \text{observe}(q) | \text{control}(u) | \text{migrate}(h) \\
\text{h} : & \quad a \mapsto v \\
\text{msg} : & \quad \langle a \leftarrow v \rangle \\
\text{req} : & \quad (a, \text{cmd})
\end{align*}
\]

Figure 3.12: Grammar for bigActors syntax.

The notation \([E \vdash b]_a\) is the same as per the actor semantics, i.e., it denotes a bigActor with a unique name \( a \in \mathcal{A} \), with local environment \( E \in \mathcal{E} \), and local behaviour \( b \). The local behaviour is specified as a sequential composition of bigActor commands \text{send}, \text{observe}, \text{control}, \text{migrate}, \text{ready}, \text{and} \text{new}, expressions \( e_{\lambda} \) that manipulate the state of the bigActor, or \text{nil}. BigActors are hosted at bigraph nodes specified by the hosting function \( \text{host} \). The notation \( \langle a \leftarrow v \rangle \) denotes a message to a bigActor \( a \in \mathcal{A} \) with content \( v \in \mathcal{V} \).

Definition 3.3 defines an execution of the BigActor model as a sequence of configurations.

Definition 3.3. A BigActor execution is a sequence \( c_0, c_1, c_2, \ldots \) where each \( c_i \xrightarrow{\lambda_i} c_{i+1} \) and \( \xrightarrow{\lambda_i} \) is derived by a semantic rule labelled \( \lambda_i \). The labels \( \lambda_i \) are the labels that name each inference rule provided by Figure 3.13.

The semantics is modelled as a transition system over the universe of bigActor configurations. The transitions are given by the inference rules presented in Figure 3.13.

The semantics can be understood as follows. A BigActor can perform actor computations. This is asserted by the rules \( \langle \text{nil}, a \rangle, \langle \text{fun} : a \rangle, \langle \text{new} : a, a' \rangle, \langle \text{snd} : a, \langle a' \leftarrow m \rangle \rangle \) and \( \langle \text{rcv} : a, m \rangle \). This rules are the similar as their actor counterparts in Figure 3.6 except \( \langle \text{snd} : a, \langle a' \leftarrow m \rangle \rangle \). The expansion of the configuration to include \( \text{host}, \eta \) and \( B \) has no
The local environment augmented so that the new actor $\langle \text{nil, a} \rangle$ significance in the rules $\langle \text{nil, a} \rangle$, $\langle \text{fun : a} \rangle$, and $\langle \text{rcv : a, \langle a \leftarrow v \rangle} \rangle$. In the rule $\langle \text{new : a, a'} \rangle$ the host map is augmented so that the new actor $a'$ will have the same host as its creator, in this case $a$. The local environment $E$ in the rules of Figure 3.13 is as the actor model.

A bigActor model executes commands $\langle \text{snd : a, \langle a' \leftarrow m \rangle} \rangle$, $\langle \text{obs : a, q, B_{obs}} \rangle$, $\langle \text{ctr : a, u} \rangle$ and $\langle \text{mgrt : a, h'} \rangle$ asynchronously. The semantics of these expressions are unique to this model and given first by the rule $\langle \text{req : a, r} \rangle$ which produces requests in $\eta$ and then by the rules labelled as $\langle \text{snd : a, \langle a' \leftarrow m \rangle} \rangle$, $\langle \text{obs : a, q, B_{obs}} \rangle$, $\langle \text{ctr : a, u} \rangle$ and $\langle \text{mgrt : a, h'} \rangle$ which consumes requests from $\eta$. The $\text{req}$ rule asserts that when any of the expressions $\langle \text{snd : a', m} \rangle$, $\langle \text{obs : q} \rangle$, $\langle \text{ctr : u} \rangle$ or $\langle \text{mgrt : h'} \rangle$ the program evolves and a tuple of the kind $(a, r)$ is added to the set $\eta$ where $a$ is the bigActor address and $r$ is the request to execute the desired com-
mand. When \( \eta \) contains a tuple of kind \((a, \text{send}(a', m))\), \((a, \text{observe}(q))\), \((a, \text{control}(u))\) or \((a, \text{migrate}(h'))\) such tuple can be consumed and the BigActor model can further advance by application of the rules \( \langle \text{snd} : a, \{a' \leftarrow m\}\rangle \), \( \langle \text{obs} : a, q, B_{\text{obs}}\rangle \), \( \langle \text{ctr} : a, u\rangle \) and \( \langle \text{mgrt} : a, h'\rangle \).

The rule \( \langle \text{snd} : a, \{a' \leftarrow m\}\rangle \) consumes a send request. The rule checks if the host of \( a \) and the host of \( a' \) are connected, i.e. if they share a link in the bigraph. If the premise is true, then a new message gets generated in the set of pending messages \( \mu \). We see the linking graph as a mean for modelling the communication infrastructure. This is the reasoning behind the premise of requesting hosts to share a link if their bigActors want to share messages.

The rule \( \langle \text{obs} : a, q, B_{\text{obs}}\rangle \) takes an observe request, interprets the respective query over the current bigraph with respect to the host, and generates a message to \( a \) inside the set of pending messages \( \mu \). The interpretation of a query into an observation of a given bigraph \( B \) relative to a host \( h \) is specified by the map:

\[
[\cdot]_h^B : \text{query} \rightarrow \mathbb{B} \quad (3.3)
\]

Queries and observations are addressed in detail in Chapter 2. The observation is then consumed as a standard message using the rule \( \langle \text{rcv} : a, m\rangle \).

The rule \( \langle \text{ctr} : a, u\rangle \) takes a control request control request, checks if the rule is local up to the host of the requesting bigActor, checks if the redex has a match in the current bigraph, and if the premisses are satisfied then it changes the bigraph replacing the redex with the reactum.

At last, the rule \( \langle \text{mgrt} : a, h'\rangle \) takes a migrate request, checks if the current host of \( a \) is connected to the desired host, and if this premise is satisfied it changes the hosting map \( \text{host}(a) \) to \( h' \). We think of migration process as sharing information between hosts about the structure of the bigActor and its current state. Thus, like in the send case, we request the host to be connected, meaning that there is some sort of communication infrastructure that can support the migration procedure.

Next we present an example of a bigActor system and its corresponding formal execution trace.

**Example 3.5.** Recall the bigraph of Figure 2.3. Consider a new app named app2 hosted at the smartphone \( sp \) that is programmed to give instructions to the user to go to street1, take a picture of the street and its buildings, then move to street3, connect to the internet and upload the picture to social2.

The bigActor system that specifies the required behaviour is given by two bigActors given in Figure 3.14.

Figure 3.15 depicts the bigActors of Figure 3.14 embedded in the bigraph of Figure 2.3.

The bigActor app2 uses the bigraph reaction rule MOVE_HOST_TO from Figure 3.11 to move from street2 to street1 and then to street3. While in street1, app2 observes the world using a query \( \text{CHILDREN}(\text{PARENT}(<\text{HOST}>)\) and saves the observation result in photo. When its host reaches street3, app2 connects to wlan0 using the bigraph reaction rule CONNECT_HOST_TO(wlan0) and finally sends photo to social2 which logs it in a local list.
val app2 = BigActor hosted_at sp with_behavior{
  control(MOVE_HOST_TO(street1))
  observe(CHILDREN(PARENT(HOST)))
  react{
    case photo =>
      control(MOVE_HOST_TO(street3))
      control(CONNECT_HOST_TO(wlan0))
      social2 ! photo
  }
}

val social2 = BigActor hosted_at srv with_behavior{
  var log = List()
  react{
    case photo => log = photo :: log
  }
}

Figure 3.14: BigActor system for scenario presented in Example 3.5.

In order to formally analyse the semantics of the bigActors of Figure 3.14, we need to encode the Scala syntax into the bigActor syntax given by the grammar in Figure 3.12. The bigActor `app2` is encoded as:

$$
[\emptyset \vdash \text{control}(r); \text{observe}(q); \text{ready}(Q); \text{control}(r'); \text{control}(r''); \text{send}(Q); \text{nil}]_{app2}
$$

where
• \( r = \text{MOVE}_\text{HOST}_\text{TO}(\text{street1}) \),
• \( r' = \text{MOVE}_\text{HOST}_\text{TO}(\text{street3}) \),
• \( r'' = \text{CONNECT}_\text{HOST}_\text{TO}(\text{wlan0}) \), and
• \( q = \text{CHILDREN}(\text{PARENT}(\text{HOST})) \).

The observation is saved in the variable \( Q \). The bigActor \text{social2} is encoded as:

\[
[\log \vdash \text{ready(\text{photo})}; e; \text{nil}]_{\text{social2}}
\]

Figure 3.16 presents an execution trace for the bigActor system presented in Figure 3.14. For the sake of succinctness we represent the behavior of \text{app2} as \( P \), the behaviour without the first command as \( P' \), the behaviour without the two first commands as \( P'' \), and so on. Likewise, we denote the behavior of \text{social2} as \( R \). We numbered each configuration to ease referencing them in the text.

Figure 3.17 depicts the trace at Figure 3.16 embedded in the respective Bigraph Reactive System. The execution trace formalized in Figure 3.16 and depicted in Figure 3.17.

The execution trace produced in Example 3.5 can now be understood as follows. The first configuration models the two bigActors \text{app2} and \text{social2} hosted at respectively \text{sp} and \text{srv}. The trace from configuration 0 to configuration 2 models the request and execution of the bigraph reaction rule \text{MOVE}_\text{HOST}_\text{TO}(\text{street1}) over the bigraph \( B_0 \) changing it to \( B_1 \). Since \text{app2} is the bigActor requesting such control action, the keyword \text{HOST} is referring to \text{sp}. Thus, the bigraph reaction rule \text{MOVE}_\text{HOST}_\text{TO}(\text{street1}) produces a new bigraph where \text{sp} moves its location from \text{street2} to \text{street1}. The execution of control commands is asynchronous, i.e. the request and the execution of the action occurs at two different time steps labelled as \text{req} and \text{ctr}. The transition labelled with \text{req} adds a request to the set of requests \( \eta \). The transition labelled with \text{ctr} consumes such request and produces the respective side-effects in the bigraph. We request the bigraph reaction rule to be local with respect of the host of the bigActor, i.e. the rule can affect the host of the bigActor and its surroundings. We address in detail local control actions in Section 2.3. The traces from 5 to 7 and 7 to 9 also model the execution of control actions requested by \text{app2}, namely \text{MOVE}_\text{HOST}_\text{TO}(\text{street3}) and \text{CONNECT}_\text{HOST}_\text{TO}(\text{wlan0}).

The trace from configuration 2 to 3 models the request of an observation command with the query \text{CHILDREN}(\text{PARENT}(\text{HOST}))\), the interpretation of such query over the current bigraph, and the generation of a message with \text{app2} (the bigActor that requested the observation) with the value of the interpretation. This is also done asynchronously, with first depositing a request at \( \eta \) using a transition labelled with \text{req} and later executing the observation over the current bigraph using the transition rule \text{obs}. \text{obs} consumes a request from \( \eta \), interprets the respective query and adds a message to the bigActor that originated the request with the value resulting from the interpretation. The interpretation of the query is also a bigraph. In this example, the interpretation of \text{CHILDREN}(\text{PARENT}(\text{HOST})) returns a bigraph with all
CHAPTER 3. BIGACTORS

0. \( \langle 0 \vdash \text{control}(r); P' \rangle_{\text{app2}}, [\log \vdash R]_{\text{social2}} \mid \text{host} \mid \emptyset \mid \emptyset \mid B_0 \)  
\( \langle \text{req.} \text{app2.} \text{control}(r) \rangle \)

1. \( \langle 0 \vdash P' \rangle_{\text{app2}}, [\log \vdash R]_{\text{social2}} \mid \text{host} \mid \emptyset \mid (\text{app2.} \text{control}(r)) \mid B_0 \)  
\( \langle \text{ctr.} \rangle \)

2. \( \langle 0 \vdash \text{observe}(q); P'' \rangle_{\text{app2}}, [\log \vdash R]_{\text{social2}} \mid \text{host} \mid \emptyset \mid \emptyset \mid B_1 \)  
\( \langle \text{req.} \text{app2.} \text{observe}(q) \rangle \)

3. \( \langle 0 \vdash P'' \rangle_{\text{app2}}, [\log \vdash R]_{\text{social2}} \mid \text{host} \mid \emptyset \mid (\text{app2.} \text{observe}(q)) \mid B_1 \)  
\( \langle \text{obs.} q, \text{host} = \text{app2} \rangle \)

4. \( \langle 0 \vdash \text{ready}(Q); P''' \rangle_{\text{app2}}, [\log \vdash R]_{\text{social2}} \mid \text{host} \mid (\text{app2} \leftarrow B_{\text{obs}}) \mid \emptyset \mid B_1 \)  
\( \langle \text{rcv.} (\text{app2} \leftarrow \text{host}) \rangle \)

5. \( \langle \text{photo} \mapsto B_{\text{obs}} \vdash \text{control}(r'); P'''' \rangle_{\text{app2}}, [\log \vdash R]_{\text{social2}} \mid \text{host} \mid \emptyset \mid B_2 \)  
\( \langle \text{req.} \text{app2.} \text{control}(r') \rangle \)

6. \( \langle \text{photo} \vdash P'''' \rangle_{\text{app2}}, [\log \vdash R]_{\text{social2}} \mid \text{host} \mid \emptyset \mid (\text{app2.} \text{control}(r')) \mid B_2 \)  
\( \langle \text{ctr.} \rangle \)

7. \( \langle \text{photo} \vdash P''''' \rangle_{\text{app2}}, [\log \vdash R]_{\text{social2}} \mid \text{host} \mid \emptyset \mid (\text{app2.} \text{control}(r'')) \mid B_2 \)  
\( \langle \text{ctr.} \rangle \)

8. \( \langle \text{photo} \vdash P'''''' \rangle_{\text{app2}}, [\log \vdash R]_{\text{social2}} \mid \text{host} \mid \emptyset \mid (\text{app2.} \text{control}(r''')) \mid B_2 \)  
\( \langle \text{ctr.} \rangle \)

9. \( \langle \text{photo} \vdash \text{send(social2.} \text{photo); nil} \rangle_{\text{app2}}, [\log \vdash R]_{\text{social2}} \mid \text{host} \mid \emptyset \mid B_3 \)  
\( \langle \text{req.} \text{app2.} \text{send(social2.} \text{photo}) \rangle \)

10. \( \langle \text{photo} \vdash \text{nil} \rangle_{\text{app2}}, [\log \vdash R]_{\text{social2}} \mid \text{host} \mid \emptyset \mid (\text{app2.} \text{send(social2.} \text{photo})) \mid B_3 \)  
\( \langle \text{end(social2 = photo)} \rangle \)

11. \( \langle \text{photo} \vdash \text{nil} \rangle_{\text{app2}}, [\log \vdash R]_{\text{social2}} \mid \text{host} \mid (\text{social2} \leftarrow \text{photo}) \mid \emptyset \mid B_3 \)  
\( \langle \text{rcv.} (\text{social2} = \text{photo}) \rangle \)

12. \( \langle \text{photo} \vdash \text{nil} \rangle_{\text{app2}}, [\log, \text{photo} \vdash \text{e} \rangle_{\text{social2}} \mid \text{host} \mid \emptyset \mid B_3 \)  
\( \langle \text{fun.} \text{social2} \rangle \)

13. \( \langle \text{photo} \vdash \text{nil} \rangle_{\text{app2}}, [\log', \text{photo} \vdash \text{nil} \rangle_{\text{social2}} \mid \text{host} \mid \emptyset \mid B_3 \)  
\( \langle \text{nil.app2} \rangle \)

14. \( \langle \text{photo} \vdash \text{nil} \rangle_{\text{social2}} \mid \text{host} \mid \emptyset \mid \emptyset \mid B_3 \)  
\( \langle \text{nil.social2} \rangle \)

15. \( \langle \emptyset \mid \text{host} \mid \emptyset \mid \emptyset \mid B_3 \rangle \)

Figure 3.16: Execution trace for the bigActor system of Figure 3.14.

the nodes inside the parent of the host. The observed bigraph is depicted in Figure 3.17
Figure 3.17: Execution trace over the respective Bigraph Reactive System.
highlighted on top of the main bigraph in the third step of the trace. Just as in the control case, observations are also specified with respect to the host of the bigActor. Observations must also be local with respect to the host of the bigActor requesting them, i.e. a bigActor can only observe the surroundings of its host. Local observations are addressed in detail in Section 2.2. The trace from 3 to 4 models app2 consuming the message with the observation and saving its value in a local variable named photo. This is done with a semantic rule labelled as rcv which is identical to the actor semantic rule with the same name.

The trace from 9 to 11 models app2 sending asynchronously a message with the value saved in photo to social2. Just like any bigActor command, the execution of sending a message also requests two steps: first req models the request of such command and later snd models the consumption of such request and the generation of a new message in the set of pending messages. The actor semantics models this as a single step depositing the message immediately to the set of pending messages µ. The main difference is that our send command involves first checking in the bigraph if the hosts of the sender and the receiver are connected. In practice this would mean to check if the bigActors are both connected by some communication protocol and infrastructure, providing means for exchanging actor messages (e.g. wireless connection, 3G, etc.). We create in η requests for any command that deals with the bigraph in some way. Thus, since sending a message demands first checking the connectivity graph in the bigraph, we create a request in η. This decision facilitates the implementation of the model since one can create a component (which we name bigraphScheduler) that is fully responsible for interacting with the bigraph. The bigraphScheduler then only needs to look for requests in η. The details on the implementation of the model are provided in Chapter 4.

The trace from 11 to 12 models a transition labelled with rcv where social2 consumes a message from the set of pending messages µ and saves the value in its local state.

The trace from 12 to 13 models an internal computation of the bigActor. This transition is labelled as fun and it plays the same role as in the actor semantics. Just as in the actor semantics, fun is left unspecified, modelling the fact that the bigActor model is agnostic to the language that is used for specify internal computation. In this case, fun is in practice is denoting the Scala command for appending photo to the Scala list log creating a new list log'.

The trace from 13 to 15 models the consumption of nil from both bigActors, which represents the end of the computation and thus the removal of the bigActors from the configuration.

The example does not illustrate the rule mgrt for the expression migrate. This rule behaves much like the others. A BigActor can request a migration by the expression migrate(h') where h' is a node in the bigraph. If the host of the BigActor and h' are connected by a link in the bigraph, the request can be consumed by the rule mgrt and the host of the BigActor will be changed. Note the bigraph remains unchanged. There is only a change in the hosting relationship between the actors and the bigraph. The link is interpreted as the physical network required for the flow of a nomadic program.

We conclude the analysis of the bigActor semantics with a few remarks.
Remark 3.1. The program in Figure 3.14, and the respective graphical representation in Figure 3.15, is concise and formal. We see the conciseness of the program and the formal interactions with a spatial model contributions of the model in the area of spatial computing.

Remark 3.2. We see the relationship between bigActors and bigraphs like the relationship between a program and its machine. Actor programs are concurrent programs and so the machine can be a distributed one. The actors model the programs and the bigraph a shared distributed machine. The programs can observe the machine (rule \texttt{obs}) and change the machine (rule \texttt{ctr}). Control can move, add, or remove nodes and do the same with connections (links). The hosting relationship is the distribution of the program over the machine. The programs can change their distribution (rule \texttt{mgrt}). The programs can exchange messages (rules \texttt{snd}, \texttt{rcv}) but only up to the machine (bigraph links). The programs can compute (rules \texttt{fun}, \texttt{term}) and create new programs (rule \texttt{new}) and locate them in the machine (locally). They could migrate thereafter. We see the BigActor model as a formal model of interactions between programs and machines and thus as a contribution to the area of bridging models for worlds with dynamic structure.

Remark 3.3. In the BigActor semantics the coupling of bigActors and bigraphs is asynchronous. Program execution puts requests in $\eta$. The requests affect the bigraph, hosting relationship, or observations at a later instant of model time. This means a bigActor can make requests executable in the bigraph at request time but unexecutable at the later time. Section 3.6 addresses this issue.

Remark 3.4. Control is local. If a BigActor seeks to replace component $R$ of a bigraph with $R'$ then the rule \texttt{ctr} requires the BigActor host to be local in $R$. This is investigated in detail in Section 2.3. We show in Section 2.2 that observation is also local. Agent-based systems embedded in spatial environments often require local observation and control. Thus, we see bigActors as suitable for agent-based modelling and programming.

Remark 3.5. The bigActor semantics follows Milner’s abstract idea of “towers of models” where different models of computation are combined by introducing just enough semantical “glue” in order to define yet another model.

In the case of the bigActor model, we picked the actor model and bigraph reactive systems and introduced an operational semantics that models interactions between the two models. This extra semantics is formalized in a way that one can look at a bigActor in a bigraphical perspective by considering solely the bigraphs of the bigActor configurations and the bigraph reaction rules that change them. Likewise, one can look at a bigActor trace from an actor perspective by abstracting out \texttt{host}, $\eta$ and the bigraph out of the bigActor configurations and all the transitions labelled as \texttt{req}, \texttt{ctr}, \texttt{obs}, and \texttt{mgrt}. Next we formalise this kind of projections over bigActor traces.

**Projections of BigActor traces**

Let $\mathcal{T}_{BA(c)}$ be the universe of bigActor traces starting with configuration $c = \langle \alpha \mid \text{host} \mid \mu \mid \eta \mid B_0 \rangle$ and let $\mathcal{T}_{Bg(K,R)}$ be the universe of bigraph traces from reaction system $Bg(K, R)$.
with signature $\mathcal{K}$ and reaction rules $\mathcal{R}$. The projection of a bigActor trace into its respective bigraph trace is provided by the map

$$projectBg : \mathcal{T}_{BA(c)} \rightarrow \mathcal{T}_{Bg(\mathcal{K}, \mathcal{R})}$$  \tag{3.4}$$

$$projectBb(c_0 \xrightarrow{l_0} c_1 \xrightarrow{l_1} \ldots \xrightarrow{l_{n-1}} c_n) \mapsto (B_0 \xrightarrow{r_0} B_1 \xrightarrow{r_1} \ldots \xrightarrow{r_{m-1}} B_m)$$  \tag{3.5}$$

where $(B_0, B_1, \ldots, B_m)$ is the sequence of different bigraphs in $(c_0, c_1, \ldots, c_n)$, and each $r_i$ represent bigraph reaction rule that creates $B_{i+1}$ due to a bigActor transition labelled as $l_j = \langle ctr : a, r_i \rangle$.

**Example 3.6.** The application of $projectBg$ to the trace of Figure 3.17 results into the bigraph trace depicted in Figure 2.7.

Let $\mathcal{T}_{Act(c_0^a)}$ be the universe of actor traces starting with the actor configuration $c_0^a = \langle \alpha_a | \mu_a \rangle$.

$$projectAct : \mathcal{T}_{BA(c)} \rightarrow \mathcal{T}_{Act(c_a)}$$  \tag{3.6}$$

$$projectAct(c_0 \xrightarrow{l_0} c_1 \xrightarrow{l_1} \ldots \xrightarrow{l_{n-1}} c_n) \mapsto (c_0^a \xrightarrow{\lambda_0} c_1^a \xrightarrow{\lambda_1} \ldots \xrightarrow{\lambda_{m-1}} c_m^a)$$  \tag{3.7}$$

where $c_i^a = \langle \alpha_i^a | \mu_i^a \rangle$ denotes an element of the sequence of actor configuration, and $\lambda_i$ denotes the elements of the sequence of actor semantics rules labels. Synthesizing the sequence of labels $\lambda_i$ is quite trivial. One must simply abstract away all the bigActor semantics labels that are either req, ctr, obs, and mgmt. We also need to remove all labels rcv that are produced due to the consumption of messages with observations. All the remaining bigActor labels (namely, fun, new, snd, rcv, and nil) represent exactly the actor semantics labels. After synthesizing the sequence of labels one must synthesize the actor configurations $c_i^a = \langle \alpha_i^a | \mu_i^a \rangle$. The sets of pending messages $\mu_i^a$ can be obtained by abstracting away all messages from in the bigActor configuration coming from observation commands. The synthesis of the sequence of actors is more complex $\alpha_i^a$ since one must produce a set of actors at $\alpha_0^a$ which execution matches exactly the derived sequence $\lambda_i$. This synthesis process is out of the scope of this work. Nevertheless, we present an example to provide the intuition behind the process.

**Example 3.7.** Consider again the bigActor system of Figure 3.14 and the respective trace of Figure 3.16. Recall the sequence of bigActor semantic labels:

$$((\text{req} : \text{app2, control(r)}), (\text{ctr} : r), (\text{req} : \text{app2, observe(q)}), (\text{obs} : q, B_{\text{obs}} = [\text{q}]^{\text{host(app2)}}_B), \newline (\text{rcv} : \langle \text{app2} \leftarrow B_{\text{obs}} \rangle), (\text{req} : \text{app2, control(r')}), (\text{ctr} : r'), (\text{req} : \text{app2, control(r'')}), (\text{ctr} : r''), \newline (\text{req} : \text{app2, send(social2, photo)}), (\text{snd} : \langle \text{social2} \leftarrow \text{photo} \rangle), (\text{rcv} : \langle \text{social2} \leftarrow \text{photo} \rangle), \newline (\text{fun} : \text{social2}), (\text{nil} : \text{app2}), (\text{nil} : \text{social2}))$$
We proceed by removing all the bigActor-specific labels obtaining the following trace:

\[
\langle \text{snd}: (\text{social2} \leftarrow \text{photo}) \rangle, \\
\langle \text{rcv}: (\text{social2} \leftarrow \text{photo}) \rangle, \\
\langle \text{fun}: \text{social2} \rangle, \\
\langle \text{nil}: \text{app2}, \langle \text{nil}: \text{social2} \rangle \rangle
\]

The behaviour that remains in the trace is the actor-specific behaviour, i.e. the concurrent asynchronous message-passing semantics. The actor system of Figure 3.18 produces such trace.

```-scala
val app2 = actor{
    social2 ! photo
}

val social2 = actor{
    var log = List()
    react{
        case photo => log = photo :: log
    }
}
```

Figure 3.18: Actor system synthesized to produce the projected trace of Figure 3.16.

Note that the behaviour specification of `social2` in Figure 3.18 is the same that its bigActor counterpart in Figure 3.14. This is expected since the bigActor version itself does not perform any bigActor-specific commands. The behaviour specification of `app2` in Figure 3.18 only contains one command, `social2 ! photo`. This is expected as well since, by looking into its bigActor counterpart in Figure 3.14, `social2 ! photo` is the only actor-specific command of `app2`.

**Scheduling policy**

The BigActor Model semantics does not force a scheduling policy for the execution of requests. Nonetheless, it is often of practical use to define a scheduling semantics. For the scope of this thesis we assume that requests are served according to a First-Come First-Served scheduling policy.

Let \( \Lambda \) denote the set of labels for the semantic rules stated in Figure 3.13. Let \( \Lambda_\eta \subseteq \Lambda \) denote the set of labels produced by the rule \( \langle \text{req}: a, r \rangle \) and \( \Lambda_\pi \) denote the set of labels produced by the rules \( \langle \text{obs}: a, q, o \rangle \), \( \langle \text{ctr}: a, u \rangle \), \( \langle \text{mgrt}: a, h' \rangle \), and \( \langle \text{snd}: a, (a' \leftarrow m) \rangle \)\). Consider a function \( \text{schd}_t : \Lambda_\pi \rightarrow \Lambda_\eta \) where, for a given execution \( t = c_0 \xrightarrow{\lambda_0} c_1 \xrightarrow{\lambda_1} \ldots \), \( \text{schd}_t(\lambda_\pi) \) returns the request consumed while executing \( \lambda_\pi \). Let \( \prec_i \subseteq \Lambda \times \Lambda \) be a total order induced by the index \( i \) of \( \lambda_i \) in \( t \).
Definition 3.4 (FCFS). An execution $t = c_0 \xrightarrow{\lambda} c_1 \xrightarrow{\lambda_1} \ldots$ follows a First-Come First-Served (FCFS) discipline if:

$$\lambda_i \prec_t \lambda_j \Rightarrow \text{schd}(\lambda_i) \prec_t \text{schd}(\lambda_j)$$

where $\lambda_i, \lambda_j \in \Lambda$. In other words, FCFS discipline requests that requests are served in the order that are requested.

3.5 Communication back-channelling

In the actor model state is local. The memory of an actor influences another only up to the messages flowing between them. In the BigActor model the Bigraph is a shared structure, and some might argue this globalizes state. For example, one may communicate between bigActors by placing and removing nodes in the bigraph and never send any inter-actor messages at all. We would consider this misuse of the model, but have nevertheless elected to provide the bigraph as a shared structure to keep the model expressive. The BigActor formalism is a model for actors operating in spatial worlds, which may in many applications be a shared space to the actors.

This misuse of the bigraph (and bigraph reaction rules) in the bigActor model can create communication back-channelling between actors. There are ways to close this channel. One may simply make the bigraph encapsulated into an actor. Then all req, obs, ctr, and mgrt actions would appear as the flow of messages between actors and the entire physical world would become the local state of one actor. This solves the problem but does not address the fundamental difficulty of coordinating observing and changing a shared machine. Therefore we have presented a semantics in which the bigraph is not encapsulated in an actor and extended the actor configuration to include explicitly the bigraph, the hosting map, and $\eta$ which models requests that affects them. By making these interactions explicit, the programmer can become aware of the programming style that we advocate with the bigActor model.

The model does support a sufficient condition, derived from its semantics for closing the bigraph communication back-channelling. One may require that two bigActors never control and observe the same region of the bigraph. The semantics of a BigActor permits us to derive the area of influence of a bigActor as expressed in Definition 3.5.

Area of influence

Communication back-channelling is defined as the explicit use of the bigraph between two bigActors for exchanging information instead of the asynchronous message passing.

Definition 3.5 (Area of influence). Let $a$ be a bigActor and $R_a = \{R_0 \rightarrow R'_0, R_1 \rightarrow R'_1, \ldots, R_n \rightarrow R'_n\}$ the set of all concrete BRRs used by it. The area of influence of $a$ over a
bigraph $B$ is the bigraph $A^B_a$ formed by the set of nodes $V_{A^B_a}$ such that $v \in V_{A^B_a} \Rightarrow \exists i. B = C \circ R_i \circ d \land v, h \in V_{R_i}$.

**Example 3.8.** Consider the bigActors app2 and social2 both hosted at respectively sp and srv. By design decision let MOVE_HOST_TO(Loc) and CONNECT_HOST_TO(wlan) be the two kinds of BRRs allowed for both bigActors.

The area of influence $A^B_{app2}$ is the bigraph formed by nodes sp, street0, street1, street2, street3, and street4 while the area of influence $A^B_{social2}$ is the bigraph formed by nodes srv, street5, and wlan2. Figure 3.19 depicts $A^B_{app2}$ and $A^B_{social2}$.

**Definition 3.6 (Reachable area of influence).** Let $A^B_a$ denote the reachable area of influence of $a$ starting at bigraph $B$, i.e. the bigraph formed by all the nodes in $A^B_{a}'$ where $B'$ is obtainable from $B$ by the the application of a finite number of rules from $R_a$.

Dually to the concept of area of influence, we define the area of observation of a bigActor as follows.

**Definition 3.7 (Area of observation).** Let $a$ be a bigActor and $Q_a = \{q_0, q_1, \ldots, q_n\}$ the set of all queries used by it. The area of observation of $a$ over a bigraph $B$ is the bigraph $O^B_a$ formed by the set of nodes $V_{O^B_a}$ such that $v \in V_{O^B_a} \Rightarrow \exists i. B_{obs} = [q_i]_{host(a)} \land v \in V_{B_{obs}}$.

**Definition 3.8 (Reachable area of observation).** Let $O^B_a$ denote the reachable area of observation of $a$ starting at bigraph $B$, i.e. the bigraph formed by all the nodes in $O^B_{a}'$ where
$B'$ is obtainable from $B$ by the the application of a finite number of rules from $\mathcal{R}_a$ or by a migration of $a$.

**Theorem 3.1.** Let $c_0 \rightarrow c_1 \rightarrow \ldots$ be an execution trace where $c_0 = \langle \alpha | \text{host} | \mu | \eta | B \rangle$ is a BigActor configuration and $\rightarrow$ is given by the BigActor semantics. Consider any two bigActors $a, a' \in \alpha$ where $a \neq a'$. If the nodes of $A^B_a$ and the nodes of $O^B_{a'}$ are disjoint sets, then $a$ and $a'$ are free from communication back-channelling.

**Proof.** By definition of communication back-channelling, two bigActors can communicate through back-channelling if one can modify the bigraph and the other can observe such modification. Thus, by requiring that the reachable area of influence of a bigActor is disjoint from the reachable area of observation of another then the two bigActors are trivially free from communication back-channelling.

**Example 3.9.** Consider two bigActors $a$ and $a'$ hosted respectively in sp and srv of Figure 3.15. Consider that such bigActors can use bigraph reaction rules of the kind \texttt{MOVE\_HOST\_TO(Loc)} and \texttt{CONNECT\_HOST\_TO(wlan)} and can observe the bigraph using the queries \texttt{HOST}, \texttt{PARENT\_HOST}, and \texttt{CHILDREN(PARENT(HOST))}. Consider that only $a$ is allowed to migrate.

Note that there are no bigraph reaction rule that let $a'$ move its host srv out of its original location, or change the bigraph in any way. Moreover, with the queries specified above, $a'$ can only observe its host srv, the parent of the host wlan2 and the nodes inside wlan2 (which is solely srv). Also note that using solely control actions, $a$ can move to any street except street5 and can connect to any wireless hotspot except wlan2. It is possible for $a$ to migrate to srv for example after moving to street3 and connect to wlan0. Nevertheless, after being hosted in srv, $a$ can not perform any of the allowed reaction rules. Thus, since $a$ can not change the structure that $a'$ can observe there are no means for communication back-channelling.

### 3.6 Correctness of BigActor Model Semantics

The BigActor model is an expressive programming model and, as such, it provides the programmer with means for writing good but also bad programs. This is common in general-purpose programming models where expressiveness is mandatory. For example, one can write in the actor model a program where an actor keeps sending messages in a loop to an actor that never receives them. This is in general considered a bad program that could in practice lead to a memory leak. Although this is still a valid behaviour up to the actor model semantics.

For the same generality argument, we do not want to constrain the bigActor model expressiveness. Every behaviour that is in the scope of the bigActor semantics is considered a valid behaviour. Nevertheless, the bigraph of a bigActor system is a shared environment and it is susceptible of issues that may lead configurations that can not progress any-more according to the bigActor semantics. We call such configurations unsafe.
A BigActor Model execution may lead to unsafe configurations due to two main issues:

- A bigActor requests a control, a migration, or a send command that can not be executed.
- Concurrency between bigActors turns executable requests unexecutable.

**Example 3.10 (Misbehaved Token Grabber).** Consider the bigActor `grabberBA` depicted at Figure 3.20 and execution depicted in Figure 3.21. `grabberBA` is hosted at `bag0` and its behaviour consists of picking tokens `t0` and `t1` and putting them inside `bag0`. The

```
val grabberBA = BigActor hosted_at bag0 with_behavior{
  control(GRAB(t0))
  control(GRAB(t1))
}
```

bigraph does not include a token `t1`. The bigActor execution trace reaches a configuration with a request `<req: grabberBA, control(GRAB(t1))>` that can not be executed. Thus, this configuration is unsafe.

**Figure 3.20: BigActor grabberBA.**

**Example 3.11 (Concurrent Token Grabbers).** Consider the bigActor system formed by the two bigActors provided at Figure 3.22. Consider now two bigActors `grabberBA0` and `grabberBA1` hosted at respectively `bag0` and `bag1`. The behavior of `grabberBA0` consists of requesting to pick the tokens `t0` and `t1`, while the behavior of `grabberBA1` attempts at grabbing `t1` and `t2`. The execution, depicted by Figure 3.23, shows `grabberBA0` grabbing `t0`, then `grabberBA1` grabs `t1`. When `grabberBA0` requests to grab `t1` the token is not available and, thus, the execution reaches an unsafe configuration. Note that if we consider the same bigActor system but with only one of the bigActors, the execution terminates in a safe configuration without any requests left. Concurrency is the reason for the execution to reach an unsafe configuration. This situation is also known as a race-condition.

**Figure 3.21: Example of an execution of grabberBA leading to an unsafe configuration.**

**Figure 3.22: Example of a BigActor system formed by two bigActors.**

We adapt the notion of progress and preservation from Benjamin Pierce’s “Types and Programming Languages” [63] to define safe bigActor configurations and present sufficient conditions for safe bigActor executions.
val grabberBA0 = BigActor hosted_at bag0 with_behavior{
  control(GRAB(t0))
  control(GRAB(t1))}
}

val grabberBA1 = BigActor hosted_at bag1 with_behavior{
  control(GRAB(t0))
  control(GRAB(t1))}
}

Figure 3.22: BigActors grabberBA0 and grabberBA1.

Figure 3.23: Example of an execution of multiple bigActors leading to an unsafe configuration.

Progress and Preservation

First we need to define a terminal configuration, i.e. a bigActor configuration where there is nothing remaining to be executed.

Definition 3.9 (Terminal Configuration). Let \( c = \langle \alpha \mid \mu \mid \eta \mid B \rangle \) be a BigActor configuration. \( c \) is terminal if \( \mu = \emptyset \), \( \eta = \emptyset \), and \( \alpha = \emptyset \) or all bigActors in \( \alpha \) are waiting to receive a message (i.e. their next command is \( \text{ready}(\cdot) \)).

Definition 3.10 (Progress). A configuration \( c \) is safe if it is terminal or it can make progress, i.e. there exists a bigActor semantics rule with label \( \lambda \) from Figure 3.13 such that \( c \xrightarrow{\lambda} c' \).
We call preservation to the property a safe configuration always transitioning to another safe configuration.

**Definition 3.11** (Preservation). Let $c$ be a safe bigActor configuration. If $c \xrightarrow{\lambda} c'$ then $c'$ is safe.

**Safety of a single bigActor**

The first step towards safety is to require that a single bigActor itself never requests operations that will lead to an unsafe configuration.

**Definition 3.12** (Feedback BigActor). Let $t = c_0 \xrightarrow{\lambda_0} c_1 \xrightarrow{\lambda_1} \ldots$ be a bigActor execution. For clarity we augment $\lambda_i$ to $\lambda_{a_i}$. $a$ is called feedback for all $r$ and $r'$ that are either control, migrate, or send the following are true:

1. $\forall j. \lambda_j = \langle \text{req} : a, r \rangle \Rightarrow \exists i < j. \lambda_i = \langle \text{obs} : a, q, B_{\text{obs}} \rangle$

2. $\forall i. \forall k. k > i \land \lambda_i = \langle \text{req} : a, r \rangle \land \lambda_k = \langle \text{req} : a, r' \rangle \Rightarrow \exists j. i < j < k \land \lambda_j = \langle \text{obs} : a, q, B_{\text{obs}} \rangle$

3. Let $B_{\text{obs}}^{a, \lambda_i}$ denote the last observation of $a$ preceding $\lambda_i$.

$$\forall i. \lambda_i = \langle \text{req} : a, r \rangle \Rightarrow \exists \lambda_i'. \langle a, \emptyset, \{r\} \mid B_{\text{obs}}^{a, \lambda_i} \rangle \xrightarrow{\lambda_i'} c'$$

where $\lambda_i'$ consumes the request $(a, r)$

The first requirement states that before any request $r$ by $a$, there must be an observation by the same bigActor. The second requirement states that between two requests $r$ and $r'$ by $a$ there must be an observation by the same bigActor. The last requirement states that, given a configuration formed by a bigraph $B_{\text{obs}}^{a, \lambda_i}$ and a single request $r$ then there is a new configuration obtained by consuming $r$. In other words, it requires that $r$ can be executed over the observation bigraph and thus also executable over the bigraph at the observation time.

**Theorem 3.2.** Consider a BigActor execution $t = c_0 \xrightarrow{\lambda_0} c_1 \xrightarrow{\lambda_1} \ldots \xrightarrow{\lambda_n} c_n$ that follows a FCFS discipline (Assumption 3.2.1). Assume that $\alpha_i$ contains a single BigActor $a$ for all $i$ and $a$ is a feedback BigActor (Assumption 3.2.2). Assume that $\eta$ and $\mu$ are empty at $c_0$ and $c_0$ is safe (Assumption 3.2.3). Then $c_n$ is safe.

**Proof.** By mathematical induction in the the number of transitions.

**Base case** Given $c_0 \xrightarrow{\lambda_0} c_1$, $c_1$ is safe.

By Assumption 3.2.3 $a$ must be busy otherwise $c_0$ is terminal (Definition 3.9) and $\xrightarrow{\lambda_0}$ can not be triggered. If $a$ is busy then it can trigger any of the following rules leading to $c_1$: fun, new, term, and req. Since the overall execution is assumed to have only one bigActor new is excluded. By Assumption 3.2.3, the rules ctr, mgrt, obs, snd can not be executed since $\eta$ is still empty. fun, and term are internal to the actor and can not generate a transition to an
unsafe configuration. By Definition 3.12 item 1, the rule req can only request an observation because a control, migrate, or a send require a previous observation.

**Inductive step** Given \( c_{i-1} \xrightarrow{\lambda_{i-1}} c_i \) where \( c_i \) is safe then \( c_i \xrightarrow{\lambda} c_{i+1} \) and \( c_{i+1} \) is safe.

Since \( c_i \) is safe then \( \lambda_i \) can be given by one of the following rules: fun, rcv, term, req, ctr, mgrt, obs, and snd (new is excluded by Assumption 2).

By the semantics presented in Figure 3.13, the rules fun, rcv, and term model internal computation of bigActors and, thus, do not produce nor consume requests that can make the configuration unsafe.

The rule req can produce a new request, i.e. \( \eta_{i+1} = \eta_i \cup \{r\} \). If \( r \) corresponds to an observation then \( c_{i+1} \) is correct since the observations do not change the bigraph. If \( r \) is either a control, a migration or a send request then it must produce a correct configuration upon \( B_{\text{obs}}^{a,r} \). By Definition 3.12, \( a \) alternates between observation and control/migrate/send then the bigraph \( B_i \) at configuration \( c_i \) can be decomposed as \( B_i = C \circ B_{\text{obs}}^{a,r} \circ d \) where \( B_{\text{obs}}^{a,r} \) is the last observation prior to \( r \). By Definition 3.12 item 3, \( r \) must be executable over \( B_{\text{obs}}^{a,r} \). Since \( B_i = C \circ B_{\text{obs}}^{a,r} \circ d \) then \( r \) also produces a correct configuration under \( B_i \).

The rules ctr, mgrt, and obs consume one request \( r \) out of \( \eta_i \). By Assumption 3.2.1, all requests of \( a \) are served in order. Since all requests in \( \eta_i \) were generated by a (single) feedback BigActor, then \( c_{i+1} \) with \( \eta_{i+1} = \eta_i \setminus \{r\} \) is safe.

Note that the proof preserves both progress and preservation for all configurations.

**Safety of concurrent bigActors**

In the multiple bigActor case, the feedback condition is not sufficient for safety. A bigActor may request an operation executable at request time, but not executable when the request is executed. This is a consequence from concurrency and asynchrony of the bigActor semantics. We take two strategies to prevent unsafe configurations due to concurrency: prevent undesired interleaving (atomic read-write semantics); prevent shared resources (partitioning semantics).

**Definition 3.13** (Atomic read-write semantics). Let \( \Lambda_{\theta} \) denote the set of all req, ctr, mgrt, and snd labels and let \( \Lambda_{\theta}^a \) be the subset of labels from \( \Lambda_{\theta} \) produced by serving requests from \( a \). An execution \( t = c_0 \xrightarrow{\lambda_0} c_1 \xrightarrow{\lambda_1} \ldots \xrightarrow{\lambda_{n-1}} c_n \) follows an atomic read-write semantics if for any \( \lambda_i = \langle \text{obs} : a_i, \cdot \rangle \) for which there exists a \( \lambda_{i+k} \in \Lambda_{\theta}^a \) then there are no \( \lambda_{i+j} \in \Lambda_{\theta}^{a'} \) where \( a \neq a' \), \( j \in \{1, \ldots, k - 1\} \), and \( k = \min\{k|\lambda_{i+k} \in \Lambda_{\theta}^a\} \).

In other words, an atomic read-write semantics requires that between any obs from \( a \) and subsequent ctr, mgrt, or snd from the same bigActor, no other ctr, mgrt, or snd can occur from another bigActor \( a' \).

**Theorem 3.3.** Consider a BigActor execution \( t = c_0 \xrightarrow{\lambda_0} c_1 \xrightarrow{\lambda_1} \ldots \xrightarrow{\lambda_{n-1}} c_n \) that follows a FCFS discipline (Assumption 3.3.1) and the execution follows an atomic read-write semantics (Ass-
sumption 3.3.2). Assume that all bigActors are feedback (Assumption 3.3.3). Assume that \( \eta \) and \( \mu \) are empty at \( c_0 \) and \( c_0 \) is safe (Assumption 3.3.4). Then \( c_n \) is safe.

**Proof.** Without loss of generality, assume that the configurations at execution \( t \) contains only two bigActors \( a \) and \( a' \) (two feedback bigActors is sufficient to lead to an unsafe configuration). By Assumption 3.3.1 and Assumption 3.3.2 the execution has no interleaving between observations and \( \text{ctr/mgrt/send} \) of two different bigActors. Thus, one can partition the sequence \( t \) into subsequences \( t_0^a, t_0' \), \( t_1, t_1' \), \( t_2' \ldots \) where each \( t_k \) denotes the execution of a single BigActor \( a \) (and possibly labels regarding rules \( \text{fun}, \text{term}, \text{new} \) and \( \text{rcv} \) which are internal to bigActors and do not lead to unsafe configurations). Since each bigActor \( a \) is feedback (Assumption 3.3.3), by Theorem 3.2, each \( t_k = c \rightarrow \ldots \rightarrow c', c' \) is safe. Thus, \( c_n \) is also safe. \( \square \)

**Definition 3.14** (Partitioning semantics). Consider an execution \( t = c_0 \xrightarrow{\lambda_0} c_1 \xrightarrow{\lambda_1} \ldots \xrightarrow{\lambda_{n-1}} c_n \). Let \( A_i^a \) denote the area of influence of \( a \) at \( c_i \), i.e., the portion of the bigraph that can be changed by means of Bigraph Reaction Rules applied with respect to the host of \( a \). An execution follows a partitioning semantics if, for every two bigActors \( a \) and \( a' \), the following holds:

1. \( \forall i. \forall j. A_i^a \cap A_j^{a'} = \emptyset \)

2. \( \forall i. \forall j. \forall v \in A_i^a. \forall v' \in A_j^{a'}. \text{Lks}(v) \cap \text{Lks}(v') = \emptyset \)

where \( \text{Lks} : V_B \rightarrow \mathcal{P}(E_B \cup Y_B) \) is a function that returns the set of links connected to ports of a given node.

In other words, partitioning semantics requires the areas of influence of any two bigActors and their links be disjoint at every time.

**Theorem 3.4.** Consider a BigActor execution \( t = c_0 \xrightarrow{\lambda_0} c_1 \xrightarrow{\lambda_1} \ldots \xrightarrow{\lambda_{n-1}} c_n \) that follows a FCFS discipline (Assumption 3.4.1) and the execution follows a partitioning semantics (Assumption 3.4.2). Assume that all bigActors are feedback (Assumption 3.4.3). Assume that \( \eta \) and \( \mu \) are empty at \( c_0 \) and \( c_0 \) is safe (Assumption 3.4.4). Then \( c_n \) is safe.

**Proof.** Without loss of generality assume the execution of two bigActors \( a \) and \( a' \). By Definition 3.14, the areas of influence and corresponding links are disjoint throughout the overall trace \( t = c_0 \xrightarrow{\lambda_0} c_1 \xrightarrow{\lambda_1} \ldots \xrightarrow{\lambda_{n-1}} c_n \). Then each \( B_i \) at configuration \( c_i \) can be decomposed into \( B_i = B_i^a \circ B_i^{a'} \) where \( B_i^a \) and \( B_i^{a'} \) are disjoint (without sharing any nodes nor edges nor names). Thus, each configuration \( c_i = \langle \{ a, a' \} | \mu_i | \eta_i | B_i \rangle \) can be decomposed into two configurations \( c_i^a = \langle \{ a \} | \mu_i^a | \eta_i^a | B_i^a \rangle \) and \( c_i^{a'} = \langle \{ a' \} | \mu_i^{a'} | \eta_i^{a'} | B_i^{a'} \rangle \) where \( \mu_i = \mu_i^a \cup \mu_i^{a'} \) and \( \eta_i = \eta_i^a \cup \eta_i^{a'} \). Since \( a \) and \( a' \) are feedback, then by Theorem 3.2, there exists two traces where \( c_0^a \rightarrow \ldots \rightarrow c_n^a \) and \( c_0^{a'} \rightarrow \ldots \rightarrow c_n^{a'} \) where \( c_n^a \) and \( c_n^{a'} \) are safe. \( c_n \) can be either \( c_n^a \) or \( c_n^{a'} \). Thus, \( c_n \) is safe. \( \square \)
3.7 Final remarks

In this chapter we introduce the BigActor model for modelling structure-aware computation, i.e., computation that interacts with its environment by sensing and controlling its location and connectivity. The model combines the Actor model [2] for modelling reactive, concurrent and distributed systems and the Bigraph model to model the structure of physical machines [3]. The BigActor Model provides a bridge between programs and machines with dynamic structure. Programs are written as actors while bigraphs describe the structure of the world in terms of the location and connectivity of computing machines.

We introduce an operational semantics for the BigActor model as an extension of the Actor model semantics. The semantics is enriched with three rules that provide means for bigActors to observe, control the bigraph, and migrate. Observations are specified using the query language introduced in Chapter 2.

We analyse the correctness of bigActor executions in terms of progress and preservation, i.e., the ability to a configuration to progress to another configuration while preserving correctness. We introduce a programming pattern for bigActors called feedback where a bigActor always observes the space that is willing to control and thus, avoiding creating control actions that could lead to unsafe configurations. We provide sufficient conditions for correct executions of feedback bigActors with the presence of concurrency. Atomic read-write semantics requests that between an observation and a control action of a feedback bigActor, no other bigActor can alter the bigraph. Partitioning semantics requests concurrent feedback bigActors to operate at disjoint areas of space.
Chapter 4

BigActor Programming Language

The BigActor Programming language (BAL) is an embedded Domain Specific Language (DSL) hosted in the Scala Programming Language [23]. The bigActor semantics is implemented as a library that extends the standard Scala Actor library. As an embedded DSL the syntax of BAL is by definition the syntax of Scala. Nonetheless, BAL uses Scala’s type system together with high-order functions and implicit conversions in order to create a domain-specific syntactic constructs.

BigActors operate over bigraphical abstractions of the world. In BAL, the bigraphical abstractions are expressed using a bigraph term language known as BGM. BigActors request observations of the underlying bigraph using a query language and request to change the bigraph using Bigraph Reaction Rules (BRRs). The execution of BRRs over the bigraph is performed using the bigraph model checker BigMC [64]. The BigActor Runtime System (BARS) is responsible for mediating the execution of bigActor instances over a bigraphical abstraction handled by BigMC.

BARS operating over BigMC provides a framework for executing bigActors over a simulated bigraphical world. The interactions with the physical world are investigated in the second part of this dissertation.

The remainder of this chapter goes as follows. Section 4.1 presents a brief introduction to the Embedded DSLs. Section 4.2 provides an overview of the Scala programming language, in particular the Actor library and the RemoteActor library. Section 4.3 introduces the BigActor Programming Language and each major component of the BigActor Runtime System (BARS). Section 4.4 presents a simulation environment for BigActors which uses the bigraphical model checker BigMC to handle the bigraphical executions. Section 4.5 presents a case study in mobile robotics, where vehicles and sensors perform a oil-spill monitoring mission specified using bigActors. We conclude in Section 4.6 with some final remarks.
4.1 What is an Embedded DSL

DSLs are programming languages that provide programmers with constructs targeting a specific application. The objective of a DSL is to facilitate the programmer to write software to a particular domain quickly and effectively, producing programs that are easy to understand and reason [65]. This contrasts with General Purpose Languages (GPLs) that aim at reaching wider communities of programmers by providing expressive language constructs that are independent of application domains.

DSLs are commonly divided into two categories: external and internal (also known as embedded). External DSLs have their own custom syntax. Building an external DSL requires the implementation of an infrastructure for scanning, parsing, compilation and/or interpretation, as well as tools for supporting the programmer such as debuggers, editors, integrated development environments, etc.

Embedded DSLs are built over a given host language (usually a General Purpose Language). The semantics of the target domain is implemented using the host language. This strategy has the advantage of not having to develop any language infrastructure. The designer of the DSL focuses mostly on the implementation of the domain semantics. The syntax of the DSL is limited to use the core syntax of the host language. There are two ways of embedding a DSL in a host language: shallow and deep. Shallow embeddings are libraries written in the host language. The library defines the abstractions, operations, and composition rules of the DSL. Deep embeddings use the host language data structures to represent DSL programs as Abstract Syntax Trees (ASTs) and provide means to interpret them.

Different host languages provide different semantic abstractions. As such, it is convenient to choose a host language with semantical constructs as close as possible to the ones desired for the embedded DSL. Haskell, Groovy, and Scala are popular host languages. They are equipped with means such as static type checking, type inference, Algebraic Data Types (ADTs), high-order functions, and implicit type conversion that can be used for achieving a domain-specific syntax without the need to construct a parser and a compiler. Ghosh [66] describes several techniques to use this languages in order to develop DSLs.

We designed BAL as a shallow embedded DSL hosted at Scala. The BigActor model is built semantically over the Actor model. Thus, we looked for hosting languages with actors as the main concurrency model. This requirement is satisfied by Scala, which uses actors as the primary concurrency model. Another factor that influenced our decision is Scala’s capabilities for hosting DSLs. Next we provide a brief overview of Scala, focusing on the capabilities that are relevant for the implementation of BAL.

4.2 Scala Programming Language

Scala is a modern programming language that runs over the Java Virtual Machine (JVM). It is a multi-paradigm language, providing the programmer with functional and object-
oriented abstractions. Scala is statically typed. However, it provides a type-inference mechanism that liberates the programmer from writing heavily type-annotated programs as in Java. The language provides an implicit type conversion mechanism. Implicit conversion is used to convert automatically the type of a value from its original type to a new type that provides additional methods or fields. Implicit conversions is used as part of a DSL construction pattern for chaining data and commands [66]. Nonetheless, this is a powerful feature and should be used with extra care since it makes types weaker.

The de facto concurrency model of Scala is the actor model. Next we provide a brief overview of the Scala Actor and RemoteActor libraries.

The Actor Model in Scala

The Actor library provides means to instantiate and invoke actors by extending the Actor trait. There are two ways of defining an actor. One can create a class that extends the Actor trait, override the act method to specify the desired behaviour, and then instantiate and invoke actor objects from the defined class. Figure 4.1 shows the definition of a simple actor class and its instantiation and invocation. The actor prints the word World as soon as it receives a message Hello. The other way to define an actor is to use the actor method defined in the Actor companion object. This provides a way to define an actor on-the-fly without the need to define a class a priori. Figure 4.2 presents the same actor of Figure 4.1 but instantiated and invoked with the actor method.

The Actor library provides commands that implement the asynchronous message-passing concurrency semantics of the actor model [2]. An actor can send a message to another actor using the “!” command. For example, a ! "Hello" sends a message "Hello" to a.

class HelloActor extends Actor{
    def act(){
        react{
            case "Hello" => println("World")
        }
    }
}

val a = new HelloActor
a.start()

Figure 4.1: A simple actor class and its instantiation and invocation.
val a = actor{
  react{
    case "Hello" => println("World")
  }
}

Figure 4.2: A simple actor instantiated and invoked using the actor method.

The react method takes a body consisting of a sequence of case statements. The messages received in an actor's mailbox are passed to the case statements to be pattern-matched. The first case to be matched by the message is the one that is executed.

The RemoteActor Library

The RemoteActor library provides methods to create and select actors that are instantiated in different machines. A remote actor needs to be identified by a unique ID of type Symbol. The symbol is unique to the JVM instance on which the remote actor is executed. A Symbol is created by pre-pending a "'" before the desired name, e.g. 'myActor.

Figure 4.3 shows a code sketch to create a remote actor. The method alive defines the

actor {
  alive(9000)
  register('myActor, self)
  // behaviour definition
}

Figure 4.3: Remote actor instantiation.

TCP port that the actor is going to use to communicate. The method register takes the unique ID symbol and the actor instance name to be registered in the remote actor system. A Symbol name can only be registered with a single actor at a time. Only when the actor is terminated, can its Symbol name be reused.

Figure 4.4 shows how to select and use a remote actor. The method select takes an

actor {
  val c = select(Node("127.0.0.1", 9000), 'myActor)
  c ! msg
  // ...
}

Figure 4.4: Remote actor selection and use.

object of kind Node and a Symbol denoting the ID of the remote actor to be selected and
returns a actor instance that can be used just as a regular actor. The Node constructor takes
the IP address of the machine where the actor is being hosted and the TCP port.

4.3 BARS - The BigActor Runtime System

In this section we discuss the architecture and implementation of BAL. BAL programs run over the BigActor Runtime System (BARS). Figure 4.5 shows the main components of BARS and their interactions. The top layer depicts the bigActor instances. The mid-

<table>
<thead>
<tr>
<th>BigActors</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOSTING_REQUEST</td>
</tr>
<tr>
<td>OBSERVATION_REQUEST</td>
</tr>
<tr>
<td>CONTROL_REQUEST</td>
</tr>
<tr>
<td>MIGRATION_REQUEST</td>
</tr>
<tr>
<td>SEND_REQUEST</td>
</tr>
<tr>
<td>HOSTING_SUCCESSFUL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BigActorSchdl</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGRAPH_REQUEST</td>
</tr>
<tr>
<td>EXECUTE_BRR</td>
</tr>
<tr>
<td>BIGRAPH_RESPONSE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BigraphManager</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGRAPH_REQUEST</td>
</tr>
</tbody>
</table>

Figure 4.5: BigActor Runtime System.

dle layer, named BigActorSchdl, represents the scheduler that takes requests from the bigActor instances and executes them. The bottom layer, named BigraphManager, represents the component that is responsible to handle the bigraphical abstraction of the world. BigraphManager takes requests from BigActorSchdl to get the latest bigraph and to execute BRRs. The architecture presented in Figure 4.5 implements the asynchronous bigActor semantics introduced in Chapter 3.

Next we explain each component and what makes this implementation faithful to the bigActor semantics.

BigActor library

The BigActor library is defined as a class named BigActor that extends the Actor trait mixed with BigActorCommands trait. Figure 4.6 presents the code skeleton of BigActor and BigActorCommands. BigActor is an abstract class with an function named behavior. This function is implemented by each bigActor in order to define its behavior. This function plays the same role as act in the Actor class definition.

BigActor overrides the function act. The implementation of act sends first a message HOSTING_REQUEST(hostID) to the BigActor Scheduler, named BigActorSchdl. If the
abstract class BigActor(hostID: Symbol) extends Actor with BigActorCommands{
  def behavior
  def act{
    bigActorSchdl ! HOSTING_REQUEST(hostID)
    receive{
      case HOSTING_SUCCESSFUL => println("BigActor successfully hosted")
    }
    behavior
  }
}

trait BigActorCommands{
  def observe(query: Query) = {
    BigActorSchdl ! OBSERVATION_REQUEST(query)
  }
  def control(brr: BRR){
    BigActorSchdl ! CONTROL_REQUEST(brr)
  }
  def migrate(newHostID: Symbol){
    BigActorSchdl ! MIGRATION_REQUEST(newHostID)
  }
  def send(msg: Message,rcv: BigActor){
    BigActorSchdl ! SEND_REQUEST(msg,rcv)
  }
}

Figure 4.6: Definition of BigActor class.

BigActor Scheduler successfully hosts the BigActor, it replies with a message HOSTING_SUCCESSFUL. After this point, act executes behavior.

The instantiation of a BigActor is similar to the instantiation of a regular actor except that we need to provide the host node to the constructor and override the method behavior instead of act. Figure 4.7 provides a skeleton of the instantiation and invocation of a BigActor.

val ba = new BigActor('hostNode){
  def behavior{
    //BigActor's behavior
  }
}

ba.start

Figure 4.7: Instantiation and invocation of a BigActor.

The BigActor library also provides a DSL pattern based on typed abstractions [66]. This
pattern provides strategies for removing as much syntactic noise as possible from the host language. For more details about implementing DSLs using implicit conversion see [66]. We use implicit conversions together with the high-order functions to provide programmers with a cleaner way of instantiating bigActors. Figure 4.8 provides the skeleton. Figure 4.8 shows

```scala
BigActor hosted_at "hostNode" with_behavior{
    //bigActor's behavior
}
```

Figure 4.8: Instantiation and invocation of a BigActor.

a distilled and readable syntax. It removes the need for parentheses. The `new` keyword is removed. Scala allows the “dot” notation to be removed while calling methods. The bigActor is invoked as soon as it is created. Thus, there is no need to execute the method `start`.

The trait `BigActorCommands` provides the implementation of the bigActor commands for observing (`observe`), controlling via BRRs to change the bigraph (`control`), changing the host (`migrate`) as well as sending messages to other bigActors (`send`). Each method sends messages to an actor named `BigActorSchdl` that implements the BigActor Scheduler. The messages are parametrized with the information required to perform the desired command. For example, the `observe(q)` sends a message `OBSERVATION_REQUEST(q)` to `BigActorSchdl`. The use of actor messages for modelling bigActor requests implements the asynchrony nature of such requests defined by the BigActor semantics.

The functions defined in `BigActorCommands` implement the semantic rule \( \langle \text{req} : \text{a}, \text{r} \rangle \) of Figure 3.13 that adds request to the set of pending requests \( \eta \). The set \( \eta \) is the modelled by the mailbox of the actor `BigActorSchdl`.

**BigActor Scheduler**

The BigActor Scheduler is modelled as an actor named `BigActorSchdl`. It takes requests from bigActors, i.e., hosting, observation, control, send, and migration, schedules and executes them upon the current bigraph. It also keeps track and updates the hosting relation. Since we are using a Scala actor to dispatch commands, the scheduling policy for the execution of bigActor commands becomes the one used in the Scala Actor library, which is First-Come First-Served (FCFS).

The interactions with `BigActorSchdl` are defined using a set of case classes that extends a trait named `BigActorSchdlAPI` (see Figure 4.9). This pattern of creating case classes that inherit from a trait or abstract class is known as Algebraic Data Types (ADT). The trait `BigActorSchdlAPI` is treated as the type constructor while all the case classes as data constructors.

Figure 4.10 provides the skeleton of the class definition for `BigActorSchdl`. `BigActorSchdl` has a local hash map named `hostRelation` that models the hosting relation between bigActors and bigraph nodes. `BigActorSchdl` has a `react` body nested in a `loop` body. The `react`
sealed trait BigActorSchdlAPI

case class HOSTING_REQUEST(hostId: Symbol) extends BigActorSchdlAPI

case class HOSTING_SUCCESSFUL extends BigActorSchdlAPI

case class OBSERVATION_REQUEST(query: String) extends BigActorSchdlAPI

case class CONTROL_REQUEST(brr: BRR) extends BigActorSchdlAPI

case class MIGRATION_REQUEST(newHostId: Symbol) extends BigActorSchdlAPI

case class SEND_REQUEST(msg: Any, rcv: BigActor) extends BigActorSchdlAPI

Figure 4.9: Algebraic data type BigActorSchdlAPI that defines a set of messages to interact with BigActorSchdl.

object BigActorSchdl extends Actor{
  val hostRelation = new HashMap[BigActor,BigraphNode]

  def act()
  loop{
    react{
      case HOSTING_REQUEST(hostID) =>
        //hosts the requester at hostID and replies back
        //if the process was successful
      case OBSERVATION_REQUEST(q) =>
        //interprets q in the current bigraph
        //sends the result back to the requester
      case CONTROL_REQUEST(brr) =>
        //executes brr in the current bigraph
      case MIGRATION_REQUEST(hostID) =>
        //migrates id to hostID
      case SEND_REQUEST(msg,receiver) =>
        //checks connectivity between the hosts of sender and receiver
        //if hosts are connected it sends msg to receiver
      case _ => println("UNKNOWN REQUEST")
    }
  }
}

Figure 4.10: Code skeleton for BigActorSchdl.

body is composed of several case alternatives. Each alternative pattern-matches upon messages defined by the ADT BigActorSchdlAPI. When a request is received, BigActorSchdl finds the corresponding alternative and executes the respective behavior.

From a bigActor semantics point of view, consuming messages from the BigActorSchdl mailbox and their respective execution of case alternatives implements the execution of requests from the set of pending requests \( \eta \). Thus, we have a case alternative for each of the following semantic rules: \( \text{obs} : a, q, o \), \( \text{ctr} : a, u \), \( \text{mgrt} : a, h' \), and \( \text{snd} : a, (a' \leftarrow m) \).

The case alternative that matches the message OBSERVATION_REQUEST(q) implements the
interpretation of a query \( q \) and sending a message to the requester with the result of the observation.

The case alternative that matches the message \texttt{CONTROL\_REQUEST(brr)} implements the execution of \( brr \) over the current bigraph (if a match exists).

The case alternative that matches the message \texttt{MIGRATE\_REQUEST(hostID)} implements the migration of the requester \texttt{bigActor} from its current host to \texttt{hostID}.

The case alternative that matches the message \texttt{SEND\_REQUEST(msg,rcv)} implements a message \texttt{msg} being asynchronously sent to \texttt{receiver} after confirming that the host of the sender and the host of \texttt{receiver} are connected in the current bigraph.

**Bigraph Manager**

In order to request observations and control actions over the bigraphical abstraction of the world, the BigActor Scheduler interacts with the Bigraph Manager, which is implemented as an actor named \texttt{BigraphManager}. The Bigraph Manager is in charge of keeping an updated bigraphical abstraction and change it upon requests from the BigActor Scheduler.

\texttt{BigraphManagerAPI} is an ADT that specifies the interactions with \texttt{BigraphManager}. See Figure 4.11 for the definition of \texttt{BigraphManagerAPI}. \texttt{BIGRAPH\_REQUEST} is a request to \texttt{BigraphManager}, which replies with a message \texttt{BIGRAPH\_RESPONSE(bigraph)} that encapsulates the current bigraphical abstraction. \texttt{EXECUTE\_BRR(brr: BRR)} is a request to execute a bigraph reaction rule \( brr \) over the current bigraph. \texttt{BigraphManager} is responsible to check if the BRR has a match over the current bigraph, and handle its execution. The dynamics of the bigraph is handled by a bigraph model checker. This is explained in detail in Section 4.4.

**Remote BigActor library**

We extended the \texttt{BigActor} library to include means for instantiating and invoking \texttt{bigActors} remotely. For this matter we use the Scala \texttt{RemoteActor} library to create the \texttt{RemoteBigActor} library.

A remote \texttt{bigActor} is registered, selected and used similarly to a remote actor (see Figure 4.12).

Figure 4.13 shows an alternative way of instantiate remote \texttt{bigActors} that handles the registering and port attribution.
CHAPTER 4. BIGACTOR PROGRAMMING LANGUAGE

//registering a remote bigActor
alive(9000)
register('myBigActor, self)

//selection and use
val c = select(Node("127.0.0.1", 9000), 'myBigActor)
c ! msg

Figure 4.12: Remote bigActor registering, selection and use.

'myBigActor hosted_at "myHost" with_behavior{
  //behavior definition
}

Figure 4.13: Remote bigActor instantiation.

The ports and IPs of bigActors are defined in a configuration file. This provides a separation of concerns between programming a bigActor program and configuring the system for its execution. We assume that the configuration file is accessible to the overall scope of the bigActor system.

In a remote bigActor system, the actors BigActorSchdl and BigraphManager are also remote actors. We assume that there are one BigActorSchdl and one BigraphManager for each remote bigActor system. A distributed solution is discussed in Chapter 6.

4.4 A simulation environment for BigActors

The Bigraph Manager serves as an interface between the world of bigActors and the world of bigraphs. Different applications require different means of constructing and handling the bigraphical abstraction. In this section we describe an infrastructure that allows the instantiation of bigActors over simulated bigraphical abstractions. This provides programmers to specify their own bigraphical abstractions for their domain and simulate the operation of bigActors over the abstraction. The simulation of the bigraph dynamics is handled by a model checker named BigMC [64]. Next we briefly describe BigMC and the integration with BARS.

The Bigraphical Model Checker - BigMC

BigMC (Bigraphical Model Checker) is a model-checker designed to operate on Bigraphical Reactive Systems. It provides a framework for designing bigraphical models and their dynamics, and check if they meet a given specification. The specification is written in predicate logic enriched with two temporal commands, predecessor and successor. The bigraphs are specified using the BGM format which follows Milner’s bigraph term language [3]. The
grammar for the BGM term language is provided in Figure 4.14. The “.” operator nests

0. `Term ::= Node.Term`
1. `Term ::= (Term)`
2. `Term ::= Term | Term`
3. `Term ::= Term || Term`
4. `Term ::= $int`
5. `Term ::= Node`
6. `Term ::= nil`
7. `Node ::= name[names]`
8. `Node ::= name`
9. `names ::= n , names`
10. `names ::= name`
11. `name ::= [a-z A-Z][a-z A-Z 0-9]*`
12. `int ::= [0-9]+`

Figure 4.14: BGM grammar.

terms, e.g. `Node0.Node1` means that `Node0` is the parent of `Node1`. The “|” operator defines siblings, e.g. `Node0|Node1` means that `Node0` and `Node1` are siblings. The “||” operator defines two terms in two different regions, e.g. `Node0||Node1` means that `Node0` is at region 0 while `Node1` is at region 1. The “$” operator is used to identify holes, e.g. `$0` is the hole number 0. The `nil` value is used as the base case for the recursive nesting of terms, e.g. `Node.nil` means that nothing is nested inside `Node`. A node is defined as `nodeID[n0,n1,...n]` where `nodeID` is the node name identifier and `[n0,n1,...n]` is the list of link names. It is also possible to define nodes without link names. A BRR is defined by the infix operator “->”, `Term0 -> Term1` denotes a BRR with redex `Term0` and reactum `Term1`.

**Example 4.1.** Figure 4.15 presents a bigraph term that models the bigraph depicted at Figure 2.3. The term language implemented in BigMC does not provide means for closing

```
street0[road].(building0 | building1) |
street1[road].(building2 | building3) |
street2[road].sp[e] |
street3[road].wlan0[internet] |
street4[road].wlan1[internet] |
street5[e0].srv[internet]
```

Figure 4.15: Term specifying the bigraph of Figure 2.3

links. Thus, all edges on the bigraph are also outer names.
BARS on BigMC

Figure 4.16 shows the architecture of BARS on BigMC. The components in orange, namely the BigActors, the BigActorSchld, and the BigraphManager, define the BigActor Programming Language environment and are the essentially the same as the ones discussed in Section 4.3.

The only difference is that the BigraphManager is now provided with a frontend to interact with a bigraph defined in BigMC using the BGM term language. This interactions are mediated by a ANTLR/Java frontend that translates bigraphs in BGM format to a Java object specified according to a Java bigraph library. The BigraphManager (which is implemented on Scala) uses the Java bigraph library instead of dealing directly with BGM terms. The frontend is a translator implemented using ANTLR\(^2\) - a framework for the implementation of lexical analysers, parsers and Abstract Syntax Tree (AST) walkers [67].

The Java Bigraph Library provides methods to call BigMC with a bigraph and a set of BRRs, fetch the current bigraph, and render a bigraph graphically in a Java Swing component.

The implementation shown in Figure 4.16 works as a stand-alone bigActor simulation environment. The model checker BigMC takes the role of bigraph execution engine. BigMC takes a bigraph and a BRR from the bigraph manager and performs reachability analysis of only one step and returns the resulting bigraph to the bigraph manager.

\(^2\) Acronym for “Another Tool for Language Recognition”.
BARS on BigMC is open source and can be obtained at the repository in the URL https://bitbucket.org/eloipereira/bigactors.

4.5 A case study in mobile robotics

We proceed with a case study in the domain of mobile robotics. The case study is adapted from the paper [68] and consists of modelling and specifying the reactive and collaborative behavior of a team of autonomous vehicles performing a bilge dumping monitoring mission.

Example 4.2. Consider the following scenario. An operator at the European Maritime Safety Agency (EMSA) headquarters detects an oil spill by analysing high-resolution optical satellite imagery and SAR imagery from the Integrated Maritime Data Environment (IMDatE) [69]. The operator suspects of a bilge dumping illegal activity by an oil tanker. In order to take legal actions, EMSA must collect evidences in the form of images and water samples of the oil-spill. To obtain such evidences the operator requests a collaborative robotic mission.

The mission is conducted by a network of three kinds of vehicles - Unmanned Aerial Vehicles (UAVs), Navy vessels, and Autonomous Underwater Vehicles (AUVs). UAVs are used for detecting and tracking oil spills as well as for collecting information of tankers that broadcast their location using Autonomous Identification System (AIS). The role of the Navy vessels is to carry AUVs to the vicinities of an oil-spill, deploy and operate them in order to take samples.

Next we explain the bigraphical abstraction for the scenario of Example 4.2, its dynamics, and the mission specification using bigActors.

Bigraphical abstraction

The signature for the logical-space model is as follows.

$$\mathcal{K} = \{\text{Loc : 0}, \text{Tanker : 1}, \text{Vessel : 2}, \text{Auv : 1}, \text{Uav : 2}, \text{HQ : 2}\} \quad (4.1)$$

The kind $\text{Loc : 0}$ is used to denote geographical spaces delimited by polygons, e.g., the space that comprises an air field, and the space that entails an oil-spill. Nodes of kind $\text{Loc}$ have arity 0, i.e., do not share any links. The kind $\text{Tanker : 1}$ is used to denote the location and connectivity of oil tankers. The connectivity of a node of kind $\text{Tanker}$ models that the respective tanker is broadcasting AIS messages. The kind $\text{Vessel : 2}$ denotes the location and connectivity of Navy vessels. Nodes of kind $\text{Vessel}$ have one link denoting connectivity to a 3G network and a second denoting connectivity to an acoustic command and control link used to operate AUVs. The kind $\text{Auv : 1}$ denotes the location and connectivity of AUVs. Nodes of kind $\text{Auv}$ have one link denoting the connectivity to an acoustic command and control link. The kind $\text{Uav : 2}$ denotes the location and connectivity of UAVs. Nodes of kind $\text{Uav}$ have one link denoting connectivity to a 3G network and a second denoting the reception
of AIS messages. The kind $\text{HQ : 2}$ denotes the location and connectivity of command and control headquarters. The connectivity of nodes of kind $\text{HQ}$ is similar to the ones of kind $\text{Uav}$.

Table 4.1 provides the sorting discipline for building bigraphs for this case study.

<table>
<thead>
<tr>
<th>Type</th>
<th>Sorting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Placing</td>
<td>$\text{Uav, Vessel, and Tanker}$ nodes can only be nested inside $\text{Loc}$ nodes</td>
</tr>
<tr>
<td>Placing</td>
<td>$\text{Auv}$ nodes can be nested inside $\text{Loc}$ and $\text{Vessel}$ nodes</td>
</tr>
<tr>
<td>Placing</td>
<td>$\text{Uav, Auv, Tanker, and HQ}$ nodes are atomic</td>
</tr>
<tr>
<td>Linking</td>
<td>$\text{Auv}$ nodes are connected to only one $\text{Vessel}$ node</td>
</tr>
</tbody>
</table>

Table 4.1: Sorting discipline for the mobile robotics case study.

Figure 4.17 presents an example of a bigraph that respects the sorting discipline presented in Table 4.1. Nodes $\text{airField}$, $\text{shore}$, and $\text{searchArea}$ are of kind $\text{Loc}$. Nodes $\text{uav0}$ and $\text{uav1}$ are of kind $\text{Uav}$. Nodes $\text{auv0}$ and $\text{auv1}$ are of kind $\text{Auv}$. Nodes $\text{vessel0}$ and $\text{vessel1}$ are of kind $\text{Vessel}$. The node $\text{emsaHQ}$ is of kind $\text{HQ}$. The BGM term that models the bigraph of Figure 4.17 is presented at Figure 4.18.

**Bigraph reaction rules**

We define two sets of bigraph reaction rules: the set of control reaction rules $\mathcal{R}_c$ and the set of environment reaction rules $\mathcal{R}_e$. $\mathcal{R}_c$ models the set of control actions that bigActors can request. $\mathcal{R}_e$ models the set of events performed by the environment. In the bigActor semantics, only bigActors can request the execution of BRRs. In order to be faithful to the semantics, we model the environment as a bigActor named $\text{env}$ that can arbitrarily request control actions from $\mathcal{R}_e$. 
searchArea.(tanker0[ais0]  
|  
spill0  
|  
uav0[3G,ais0]  
|  
auv0[a0]  
|  
vessel0[3G,a0])  
|  
shore.(vessel1[3G,a1].(auv1[a1]))  
|  
airfield.(uav1[3G,ais1])  
|  
emsaHQ[3G,ais2]

Figure 4.18: BGM term for the bigraph depicted in Figure 4.17.

Using the Discrete Event Systems terminology, one can interpret the transitions triggered by rules on $R_c$ as the set of controllable events while transitions triggered by rules from $R_e$ as the set of uncontrollable events [70, 71].

Figure 4.19 presents a set of abstract BRRs modelling $R_c$. The rule MOVE_HOST_TO(loc) models a vehicle moving from its current location to the location specified by loc. The term HOST in the rule is replaced by the host of the bigActor that requests the rule. The kind of HOST for this particular rule is either UAV or Vessel. The rule DEPLOY(auv) models a vessel deploying auv. The term HOST in the rule is replaced by the host of the bigActor, which must be of kind Vessel. The rule COLLECT(auv) is the dual of the rule DEPLOY(auv). The rule SAMPLE(auv,oil) models a vessel requesting auv to take a sample of oil. The sample is a node nested inside auv with a name oilSample. The term HOST in the rule is replaced by the host of the bigActor, which must be of kind Vessel.

Figure 4.20 presents a set of abstract BRRs modelling $R_e$. The rule CONNECT_TO(link,x) models a link x being connected to link. The rule DISCONNECT_FROM(link,x) is the dual of the rule CONNECT_TO(link,x). The rule NEW_OIL(oil,loc) models a new node of kind Loc named oil being generated inside the node loc. The rule NEW_TANKER(tanker,loc) is similar to NEW_OIL(oil,loc) excepts that it generates a new node of kind Tanker named tanker.

**Mission specification**

The bigActor provided in Figure 4.21 specifies the oil-spill monitoring mission. The bigActor is hosted at emsaHQ. It starts by requestion an observation with query LINKED_TO(HOST). This observation models the operator at EMSA accessing which resources, i.e., UAVs and
Figure 4.19: Abstract BRRs modelling the set of control actions.

Figure 4.20: Abstract BRRs modelling the set of environment actions.
BigActor hosted at "emsaHQ" with_behavior{
    observe(LINKED_TO(HOST))
    react{
        case obs =>
            val uavs = obs.filter(isUav)
            val vessels = obs.filter(isVessel)
            if(!uavs.isEmpty & !vessels.isEmpty){
                BigActor hosted_at uavs[0] with_behavior track_oilSpills(vessels[0])
                BigActor hosted_at uavs[0] with_behavior ais_receiver(this)
                val log = new MutableList[Any]
                loop{
                    react{
                        case msg => log += msg
                    }
                }
                else log.error("Not enough resources.")
            }
    }
}

Figure 4.21: BigActor specifying the oil-spill monitoring mission.

vessels, she has available. When the observation arrives, it is assigned to the local variable obs. obs is filtered in order to select the available UAVs and vessels. In case obs does not contain at least one UAV and one vessel, the bigActor returns an error. Otherwise it selects the first UAV and vessel to perform the mission. Next, two bigActors are created, both hosted at the first UAV from the list of observed UAVs. The first bigActor has a behavior specified by the function track_oilSpills described at Figure 4.22. The behavior track_oilSpills takes as argument a bigraph node denoting the vessel selected to perform the mission. It requests the host of the bigActor, i.e., the UAV, to move to the location named searchArea. Then, in a loop, the bigActor performs an observation with query CHILDREN(PARENT(HOST)). Observations are bounded to the variable obs. For each oil-spill found in obs, the bigActor requests a control action to move its host to the location of the oil-spill. Next, it creates a new bigActor hosted at vessel with behavior specified by the function sample_spill described at Figure 4.23. The behavior sample_spill performs an observation in order to find which AUVs are inside the vessel. If there is at least an AUV, the bigActor takes the first one, requests to deploy it, sample the oil-spill, and collects it back. Otherwise, the bigActor returns an error.

The other bigActor spawned by the bigActor described in Figure 4.21 is hosted at the UAV with behavior specified by the function ais_receiver described at Figure 4.24. This bigActor models the AIS receiver. It observes in a loop the connectivity of its host using the query LINKED_TO(HOST) and, in case the observation contains any tanker, sends the observation to the bigActor address rcv passed as argument. Note that in the bigActor of
def track_oilSpills(vessel: BigraphNode) = {
  control(MOVE_HOST_TO(searchArea))
  loop{
    observe(CHILDREN(PARENT(HOST)))
    react{
      case obs =>
        val spills = obs.filter(isOilSpill)
        spills.foreach{spill =>
          control(MOVE_HOST_TO(spill))
          BigActor hosted_at vessel with_behavior sample_spill(spill)
        }
    }
  }
}

Figure 4.22: Definition of track_oilSpills.

def sample_spill(spill: BigraphNode) = {
  observe(CHILDREN(PARENT(HOST)))
  react{
    case obs =>
      val auvs = obs.filter(isAUV)
      if (!auv.isEmpty){
        control(DEPLOY(auv[0]))
        control(SAMPLE(spill))
        control(COLLECT(auv[0]))
      }
      else log.error("Not enough resources.")
  }
}

Figure 4.23: Definition of sample_spill.

Figure 4.21, the argument rcv is assigned to the bigActor that originates the mission. This bigActor receives all messages and logs them in a logger named log.

Figure 4.25 depicts the timeline for an execution of this bigActor system. On the left-hand side of Figure 4.25 one can see the timelines for the four bigActors created throughout the execution plus the timeline of the BigActor Scheduler and the Bigraph Driver. On the right-hand side we depict the bigraphical execution. Time flows downward. We explicitly represent all interactions using arrows labelled with the messages exchanged between bigActors, BigActor Scheduler, and Bigraph Manager. In order to make the diagram less cluttered we use different colors for the interactions of each bigActor. The only action from the environment in this particular example is the ais connection established between uav0
def ais_receiver(rcv: BigActor) = {
  loop{
    observe(LINKED_TO(HOST))
    react{
      case obs =>
        val tankers = obs.filter(isTanker)
        if(!tankers.isEmpty)
          rcv ! tankers
    }
  }
}

Figure 4.24: Definition of ais_receiver.

and tanker0. For the sake of space we do not explicitly show this interaction in the diagram.

4.6 Final remarks

In this chapter we present an implementation of the BigActor model introduced in Chapter 3. The BigActor Programming Language (BAL) is a spatial programming language, where programs and written as actors [2], while the space is described as bigraphs [3]. BAL is developed as a Embedded DSL over the Scala Programming Language. Scala is a function/object-oriented language that provides actors as its main concurrency model. It is also a language with constructs that eases the development of Embedded DSLs.

In BAL, a programmer can specify bigActors that can perform the regular actor commands, i.e., send messages asynchronously to another bigActor and spawn new bigActors. Additionally, a bigActor can interact with the spatial environment modelled as a bigraph by requesting local observations and control actions.

Space is modelled symbolically as bigraphs. Any physical entity modelled as a bigraph node, e.g., a vehicle or a sensor, is treated logically as a location. Thus, by performing spatial observations, a bigActor can observe which vehicles and sensors are available and use them to instantiate new bigActors. This technique is used in the bigActor depicted in Figure 4.21 in order to select a UAV and a vessel to perform the oil-spill mission.

The query language is extended to query not only the bigraph but also the hosting relation. For example, a bigActor to query for other bigActor names hosted at specific locations. This provides means to model clusters of bigActors. Consider the command broadcast defined in Figure 4.26. This command uses a query HOSTED_AT(LINKED_TO(HOST)) to send a message to all the nodes linked to the host of the bigActor executing the command. In the pure actor model, an actor can only know the address of another actor by exchanging messages. A broadcasting command using pure actors would require the programmer to handle the propagation of address information throughout the actor system.
Figure 4.25: Timeline for an execution of the BigActor system that specifies the oil-spill monitoring example.
def broadcast(msg: Any) = {
    observe(HOSTED_AT(LINKED_TO(HOST)))
    react{
        case obs =>
            obs.foreach{b =>
                b ! msg
            }
    }
}

Figure 4.26: Definition of \texttt{broadcast} command that broadcasts a message to all bigActors hosted at nodes linked to the host of the bigActors executing the command.

Querying the hosting relation also provides a way of observing if a bigActor is still alive. For example, a bigActor may stop because of a machine failure. One can define a supervisory bigActor that persistently observes the hosting relation and spawn new bigActors at run-time whenever a given bigActor dies.

The evolution of the bigraph as a spatial model is made in discrete, event-driven time. Time advances due to the execution of BRRs. One could benefit from real-time abstractions in order to specify actions that would only happen at a specific time, e.g., search a given area for oil-spill during 2 hours. This can be achieved by extending the bigActor semantics with time constraints in similar way that the actor model is extended by Nielsen and Agha [61]. Combining the bigActor model with a new model of time follows Milner’s view of “Tower of Models” [20] where different models of computation are combined semantically. Nonetheless, this extension is not in the scope of this dissertation and remains as future work.
Part II

Bridging logical and physical machines
Chapter 5

Logical-Space Computing

5.1 Introduction

Computation is becoming ubiquitously and spatially embedded in our environment. Mobile cyber-physical systems such as smartphones and robots are equipped with sensors and actuators that observe and manipulate their spatial environment. This kind of computation that exhibits a behavior in space is commonly known as spatial computing [12]. In this thesis we call spatial programming to the practice of specifying spatial computing behaviours and spatial programming language to any infrastructure or framework that provides means for spatial programming.

Arguably, the common practice of spatial programming is to define the behavior of entities in a geometrical coordinate system, such as GPS coordinates and indoor local coordinates [72]. Geometrical coordinates result from a direct measure of physical properties, e.g., a GPS location, commonly known as latitude and longitude, is defined as the angular distance from, respectively, the equator and the Greenwich meridian. Thus, we call physical-space programming to the practice of programming using geometrical coordinates.

Example 5.1. Consider the example of an Unmanned Aerial Vehicle (UAV) collecting imagery of an oil-spill due to a suspicious illegal bilge dumping activity by an oil tanker. The exact location of the oil-spill is unknown a priori, although, due to Automatic Identification System (AIS) information collected from the tanker, it is known to be within a given rectangular area parametrized by its North-East and South-West GPS locations, (37.04, −8.59) and (36.94, −8.79). The UAV operator performs the following steps: select an UAV to perform the mission; specify a searching pattern comprised by the sequence of GPS locations to be visited inside the suspected area; as soon as the operator gets the information of the oil-spill location by some source, specify a new location to be visited. The mission is specified as a sequence of waypoints using a given format such as the Waypoint File Format (WFF) of the Mavlink Waypoint Protocol (MWP) [73]. Figure 5.1 depicts the physical-space execution of this example.
CHAPTER 5. LOGICAL-SPACE COMPUTING

Figure 5.1: Physical-space execution for Example 5.1.

Programming over physical-space gives the programmer with a large expressiveness over the spatial behavior. The UAV operator of Example 5.1 can send the UAV to any GPS location that UAV is physically capable of executing. This large expressiveness puts the responsibility for the correct specification of spatial behaviors in the hands of the programmer. For example, a simple mistake in the specification of the GPS coordinates of the UAV waypoints can lead to a completely unexpected behavior. Physical-space models also do not model explicitly relations between spaces. For example, it would be important for the UAV operator to know if the UAV is at the search area without the need to perform any further calculations.

The literature presents several programming models that approaches spatial computing from a physical level, such as Amorphous Computing [74], Spatial Programming [44], and the framework Gaia [46].

Another common approach on spatial computing is to model space symbolically, where locations are defined as symbols and explicit relations over those symbols [72]. An everyday example of a symbolic location model is a mailing address system, which specifies locations using house numbering, street names, city names, and postal codes. Mailing address systems provide an explicit containment relation between locations. For example, the US Postal address “920 Grizzly Peak Boulevard, Berkeley, CA” tells us that the house denoted by “920” is physically at the street named “Grizzly Peak Boulevard” which is physically located in the city of “Berkeley” and physically contained by the state of “CA”.

In contrast to physical-space programming, we call symbolic-space programming to the practice of programming using symbolic-space models. One of the pioneer symbolic-space programming models is the Ambient Calculus by Luca Cardelli [18]. A mobile ambient is a computing entity that can only compute at bounded locations and is allowed to communicate to other mobile ambients that share its location. In Ambient Calculus locations form a tree-like structure.

Robin Milner introduced the Bigraphical model [3] that combines a nested location model with a model of connectivity. With Bigraphs, Milner combines the tree-like location model of the Ambient Calculus with the dynamic connectivity that he introduced in the $\pi$-calculus. A bigraph changes to another bigraph upon the application of a Bigraph Reaction Rule (BRR).

Example 5.2. Recall the Example 5.1. Figure 5.2 shows a symbolic-space execution for the oil-spill monitoring mission using bigraphs. The execution results from the application of
Symbolical-space programming provides an abstract spatial model with explicit relations between locations. These models are in general convenient for specifying and formally verifying high-level spatial behaviours. However, they are oblivious to information from the physical space, which is necessary to specify the behaviour of the physical machines. For example, a conventional autopilot that equips a UAV requests waypoints to be defined by their GPS coordinates. Physical time is also not considered in a logical-space model. For example, in a symbolic model, the action of moving an UAV from its current location to another location is executed as soon as it is requested. In reality, this control action does not execute instantaneously. It might not even execute at all due to some adversarial condition from the environment.

To overcome the lack of physical information on symbolic space models, one can find in the literature symbolic models that are augmented with physical information. These models are commonly known as hybrid location models. Jiang and Steenkiste [75] introduce Aura Location Identifiers (ALI) that combine hierarchical symbolic locations with physical coordinates. Recall the the Example 5.1. The UAV location when it is over the oil-spill can be specified...
as the following Aura Location Identifier: ali://sea/searchArea/oilSpill#(36.99,-8.69).

Hybrid-space models provide means for synthesizing control actions that can be deployed in the physical machines. However, the programmer is now in charge of handling both physical and symbolic information and can introduce inconsistencies between logical and physical model. For example, the ALI ali://sea/searchArea/oilSpill#(36.99,-8.69) is inconsistent if the physical location of the oil-spill does not contain the GPS location (36.99, -8.69).

In this chapter we introduce logical-space computing. In logical space computing, the programmer manipulates a symbolical abstraction of the world, named the logical-space model, while the runtime system is in charge of manipulating a physical abstraction of the world, named the physical-space model. Both abstractions are loosely coupled by the logical-space semantics, which provides an asynchronous semantics for the execution of control actions and for the observation of the structure.

The remainder of this chapter goes as follows. We provide define physical spaces, and logical spaces, and introduce a mathematical formalism to bind these two models together called spatial structures. Next we introduce the formal semantics of logical-space computing as spatial agents that operate over spatial structures. Spatial agents are embedded in the logical model and are able to observe and request control actions. The logical-space computing semantics specifies the interactions with the physical model. We proceed by showing how the BigActor model [1] can be used for logical-space programming and we conclude with an analysis of the correctness of logical-space behaviours.

5.2 Spatial Structures

We define a logical model as a set of logical places, i.e., symbols, and a set of relations between logical places, e.g., parenting relation.

**Definition 5.1** (Logical-space model). A logical-space model $L$ is an algebraic structure with domain $\text{dom}(L)$ denoting a set of symbolic locations (names) and $n_L$ relations $\sigma_L = (\text{rel}_L^0, \text{rel}_L^1, \ldots, \text{rel}_L^{n_L-1})$, where $\text{rel}_L^i$ is a symbol denoting a relation over $\text{dom}(L)$ with arity provided by $\text{ar}_L : \sigma_L \to \mathbb{N}^1$.

A physical model is defined similarly. A physical model has a set of physical places, e.g., a set of polygons defined using some geometrical coordinates, and a set of relations between physical places, e.g., containment relation of polygons.

**Definition 5.2** (Physical-space model). A physical-space model $P$ is a mathematical structure with domain $\text{dom}(P)$ denoting a set of physical locations, e.g. polygons defined using GPS coordinates, and $n_P$ relations $\sigma_P = (\text{rel}_P^0, \text{rel}_P^1, \ldots, \text{rel}_P^{n_P-1})$ where $\text{rel}_P^i$ is a relation over $\text{dom}(P)$ with arity provided by $\text{ar}_P : \sigma_P \to \mathbb{N}$.

---

For convenience of notation we overload the relation symbols $\text{rel}_L^i$ to also denote their interpretation.
A spatial structure binds these two abstractions together using two maps: an interpretation of logical locations in physical locations, e.g., the logical name “Berkeley” is physically interpreted as the polygon that defines the city limits, and an interpretation of logical relations in physical relations, e.g., the relation between “Berkeley” and “California” is physically interpreted as the containment relation between the two polygons that define both the city and the state.

Definition 5.3 (Spatial structure). A spatial structure is a tuple \((L, P, \beta)\) where:

1. \(L\) is a logical-space model,
2. \(P\) is a physical-space model,
3. \(\beta : \text{dom}(L) \hookrightarrow \text{dom}(P)^2\) is the physical interpretation of logical locations, and
4. for each logical relation \(\text{rel}^L_i\) defined in \(\sigma_L\) there is a corresponding physical relation \(\text{rel}^P_i\) defined in \(\sigma_P\), i.e., there is a one-to-one correspondence between logical relations and physical relations.

5.3 Bigraphs and Polygons

Consider a spatial structure \((B, P, \beta)\), where \(B\) is a bigraph and \(P\) is a physical model composed of polygons and relations over polygons. The domain of \(B\) is a set of bigraph nodes and a set of link names and \(\sigma_B = (\text{parent}_B, \text{linking}_B)^3\). The relation \(\text{parent}\) defines the placing structure of a bigraph as a tree, i.e., \((n', n) \in \text{parent}_B\) means that \(n'\) is the parent of \(n\) where both \(n\) and \(n'\) are bigraph nodes. The relation \(\text{linking}\) defines the linking structure as an hypergraph, i.e., \((\text{link}, n_0, n_1, \ldots, n_j) \in \text{linking}\) means that the bigraph nodes \(n_0, n_1, \ldots, n_j\) are connected through a link named \(\text{link}\). Note that the link names must be included in the domain of the logical-space model. This is necessary since link names are part of bigraph support. Starting with \(\pi\)-calculus [17] and later with Bigraphs [3] treat channels and links as means of mobility. As such, they are treated abstractly as logical locations. See Section 2.1 and [17, 3] for further details.

The domain of \(P\) is a set of polygons and physical connections defined as \(\text{dom}(P) = \{\beta(n) \mid n \in \text{dom}(B), \beta(n) \text{ is defined}\}\) and \(\sigma_P = (\text{parent}_P, \text{linking}_P)\). Physical locations are defined as tuples \((\text{name}, \text{poly})\) where the first element of the tuple is a unique name\(^4\) and the second element is a sequence of geometrically defined coordinates defining a polygon. We also admit polygons with only one coordinate, i.e., a point.

The relation \(\text{parent}_P\) is the physical counterpart of the bigraph parenting relation denoted as \(\text{parent}_B\), while \(\text{linking}_P\) is the physical counterpart of the bigraph linking relation denoted

\(^2\)The notation \(f : A \hookrightarrow B\) denotes a partial function \(f\) with domain \(A\) and co-domain \(B\).

\(^3\)The original definition of Bigraphs by Milner [3] formalizes the parenting and connectivity of nodes using functions. For the sake of a more general approach we treat the parenting and linking as relations.

\(^4\)For the sake of convenience, we assume that polygons take the same named as their logical counterparts.
as $linking_B$. In order to cope with Milner’s bigraph definition we must enforce that the resulting parenting relation forms a tree. This limitation can be removed by using Sevegnani’s Bigraphs with Sharing [76] but this is outside the scope of this thesis. Thus, the physical counterpart of the parenting relation must also form trees. A polygon cannot be partially contained on another polygon, otherwise, the resulting parenting relation may form a cycle.

**Example 5.3.** Figure 5.4 shows an example of a spatial structure using bigraphs and polygons as, respectively, logical and physical space models. There are three polygons denoted in Figure 5.4, i.e., $(\text{searchArea}_0, ((37.04, -8.59), (37.04, -8.79), (36.94, -8.79), (36.94, -8.59)))$, $(\text{shore}, ((37.17, -8.61), (37.17, -8.56), (37.11, -8.56), (37.11, -8.61)))$, and $(\text{uav}_0, (36.97, -8.64))$.

We allow the physical interpretation $\beta$ to be a partial function, i.e., there might exist some $l \in \text{dom}(l)$ where $\beta(l)$ is undefined. This design decision has an important practical implication. One can model logical locations that get their physical interpretation later in the execution or that are purely logical and do not have a physical interpretation at all.

**Example 5.4.** Recall the Example 5.1. Consider that the logical location $\text{oilSpill}_0$ is known to be contained by $\text{searchArea}_0$ but its physical location is initially unknown. During the execution, sensors equipping the UAV detect and georeference the oil-spill, providing a physical interpretation to $\text{oilSpill}_0$. Figure 5.5 shows the logical-space execution for Example 5.1. The physical interpretation of the execution depicted in Figure 5.5 is provided in Table 5.1.

The names of physical locations remain invariant during execution while the physical relations may change over time. For example, $\text{uav}_0$ is physically contained by $\text{shore}$ in the
Figure 5.5: Logical-space execution for Example 5.1.

first step of the execution depicted by Figure 5.5, i.e., \((uav0, shore) \in parent_P\), but this is false for the remaining steps.

Note that \(oilSpill0\) does not have a physical interpretation in the beginning of the execution, i.e., \(\beta_0(oilSpill0)\) is undefined. Nonetheless, the parent of \(oilSpill0, searchArea0\), has a physical interpretation \(\beta_0(searchArea0)\). Thus, there is enough physical information to generate a searching pattern inside the polygon of the corresponding physical interpretation of \(searchArea0\). Later, when \(oilSpill0\) acquires a physical interpretation, i.e., \(\beta_2(searchArea0)\), one can specify a new waypoint inside the respective polygon. This iterative execution using the parenting relation between logical locations is later formalized in Section 5.5.

The linking graph of bigraphs can be used for defining different kinds of connectivity between entities, e.g., communication links and physical adjacency. We primarily use the linking graph to model the communication infrastructure between physical computing entities. The linking graph is defined as a hypergraph, i.e., each link can have an zero or more nodes. In concrete bigraphs, links are named by a support. Let \(\mathcal{N}\) be the set of link names. In order to incorporate the support in our logical model we assume that the link names are locations, i.e., \(\mathcal{N} \subset \text{dom}(L)\). Without lost of generality, the linking graph is modelled by the binary relation \(\text{linking}_L \subseteq \mathcal{N} \times (\text{dom}(L) \setminus \mathcal{N})\). For example, consider two hyperlinks named \(wlan0\) and \(lan0\) where \(wlan0\) connects \(smartphone0, tablet0\), and \(lan0\) connects \(pc0\). Then, \(\text{linking}_L = \{(wlan0, smartphone0), (wlan0, tablet0), (lan0, pc0)\}\). The physical counterpart is named \(\text{linking}_P\).

Example 5.5. UAVs are often controlled by more than one Control Station (CS) in the same mission. For example, an UAV can get under the control authority of a CS installed in a vessel in order to improve its range. The manoeuvre of changing control authority of an UAV from one CS to another is known as handover. For UAVs equipped with Piccolo autopilots [77], the handover manoeuvre goes as follows. Consider that an UAV is under the
control authority of Alice at CS A using the communication channel 0. Alice wants to hand 
the control authority to Bob at CS B onboard of a navy vessel.

1. Bob sets CS B to listen to channel 1 and reports by radio to Alice that he is ready for 
handover.

2. Alice at CS A changes the UAV communication channel from channel 0 to channel 1. 
At this point, Alice looses communication with the UAV.

3. As soon as the UAV establishes communication with CS B, Bob assumes control au-
thority and declares a successful handover to Alice, which turns off his communication 
link

4. If the communication is not established before a given timeout, the UAV changes 
automatically its channel back to 0 and returns to control authority of Alice.

We consider five logical locations, i.e., shore, uav0, searchArea0, csA, and csB. The logical 
locations are interpreted in polygons just as per Example 5.4. We consider two logical 
connections named ch0 and ch1 that are interpreted in two physical communication channels 
(channel0 and channel1) of the Piccolo communication protocol.

The logical-space execution is depicted in Figure 5.6. We physically depict the communi-
cation channels by a circle surrounding the respective CS. The UAV is depicted as a triangle and assumes the color of the communication channel in use, or black if no communication link is established. Note that ch1 does not have initially a physical interpretation. Nonetheless, it is known that Bob uses this channel and, as such, there is a logical link between csB and ch1. As soon as Bob turns channel1 on, ch1 becomes physically interpreted. When Alice changes the UAV to channel1, even though she looses communication, the UAV remains logically under her command since if the manoeuvre fails, the UAV returns to channel0. As such, the connectivity of uav0 in the logical model only changes to ch1 when the manoeuvre is succeeded.

5.4 Consistency of spatial structures

A spatial structure establishes a map between a logical model and a physical model. Consistency is a property about the correctness of this map.

Definition 5.4 (Consistency). A spatial structure \((L, P, \beta)\) is consistent if for each relation \(rel^L_i\) defined in \(\sigma_L\):

\[
\forall (l_0, \ldots, l_{n-1}) \in rel^L_i; p_0 = \beta(l_0) \land \ldots \land p_{n-1} = \beta(l_{n-1}) \Rightarrow (p_0, \ldots, p_{n-1}) \in rel^P_i
\]

(5.1)

where \(n = \text{ar}_L(\text{rel}_i)\). In other words, a structure \((L, P, \beta)\) is consistent if the map \(\beta\) between logical locations and physical locations preserves the relations.

Note that if a given logical location does not have a physical interpretation then consistency is vacuously true.

Example 5.6. Figure 5.7 depicts an inconsistent structure. Note that the parent of uav0 in the logical model is shore while at the physical level, the physical interpretation of uav0 is contained by the physical interpretation of searchArea0.
Consistency is a strong property for spatial structures. However, it can be expensive to ensure since it requires checking the physical interpretation of the overall logical model. It is often sufficient to check if a structure is locally consistent with respect to a given region of interest of the logical model.

In order to define local consistency one must first define what it means for a logical model to be contained by another.

**Definition 5.5 (Logical containment).** We define the logical containment as a binary relation over logical-space models and denote it by ⊆_L. Let L and L’ be two logical-space models. L ⊆_L L’ iff dom(L) ⊆ dom(L’) and ∀i.(l₀, l₁, ..., l_{n-1}) ∈ rel^L_i ⇒ (l₀, l₁, ..., l_{n-1}) ∈ rel^{L’}_i.

We define a similar relation ⊆_P for physical spaces.

**Definition 5.6 (Physical containment).** We define the physical containment as a binary relation over physical-space models and denote it by ⊆_P. Let P and P’ be two logical-space models. P ⊆_P P’ iff dom(P) ⊆ dom(P’) and ∀i.(l₀, l₁, ..., l_{n-1}) ∈ rel^P_i ⇒ (l₀, l₁, ..., l_{n-1}) ∈ rel^{P’}_i.

**Definition 5.7 (Local Consistency).** Let S = (L, P, β) be a spatial structure. S is locally consistent with respect to L’ if L’ ⊆_L L and (L’, P, β) is consistent.

Local consistency is a weaker property that is necessary and sufficient for a sound logical space computation. Next we define the semantics of logical-space computing and formally prove this statement.
5.5 Logical-Space Computing Semantics

We formalize logical-space computing semantics as a set of spatial agents that operate over a spatial structure. Spatial agents are computing entities with local state and local behavior. Their definition follows the same notation of Actors and BigActors introduced in Chapter 3.

**Definition 5.8 (Spatial Agent).** A spatial agent is a computing entity denoted as $[E \vdash b]_a$ where $a$ is a unique identifier, $E$ is the local state, and $b$ is the local behavior. The behavior of the agent is specified by some host language enriched with syntax and semantics for spatial interaction. The syntax is defined by the following grammar:

$$
\begin{align*}
b &:= \exp(;\exp)^* \\
\exp &:= f | \text{observe}(q) | \text{react}(x) | \text{control}(r)
\end{align*}
$$

where $f$ denotes the core syntax of a given host language, $q$ is a query in some query language $Q$, $x$ is a local variable over the set of variables $X$, $r \in L \times L$ is a spatial reaction rule denoted as $R \Rightarrow R'$ where $R$ is a logical-model called the redex and $R'$ is a logical-model called the reactum. The application of $R \Rightarrow R'$ over a model $L$ replaces $R$ with $R'$ if $R \subseteq L$ producing a new model $L'$ where $R' \subseteq L'$.

In other words, the behavior of a spatial agent is a sequence of commands of the kind:

- $f$ denotes abstractly any internal computation performed by a given host language.
- $\text{observe}(q)$ requests an observation of the logical model specified by a query $q$.
- $\text{react}(x)$ assigns to a local variable $x$ the value of a requested observation.
- $\text{control}(r)$ requests the execution of a control action over the logical model specified by a spatial reaction rule $r = R \Rightarrow R'$.

The purpose of the grammar is to introduce syntactic means to define the semantics of spatial agents. The behavior of spatial agents can be specified by different kinds of host languages with their own syntactic constructs. In this dissertation, we use Scala as the host language enriched with the BigActors commands described in Chapter 4.

The logical-space computing semantics is modelled as a transition system over spatial-computing configurations.

**Definition 5.9 (Spatial Computing Configuration).** A configuration is a tuple $\langle \alpha \mid S \mid \eta \rangle$ where:

- $\alpha$ is a finite set of spatial agents,
- $S$ is a set of spatial configurations,
- $\eta$ is a set of initial spatial configurations.

Spatial reaction rules is an adaptation of the Bigraph Reaction Rule introduced by Milner [3] to logical-space models.
CHAPTER 5. LOGICAL-SPACE COMPUTING

- $S$ is a spatial structure, and
- $\eta$ is a finite set of requests.

The requests in $\eta$ are of the following kind:

- $\text{OBS}(a, q)$ is a request from agent $a \in \alpha$ for an observation specified by the query $q \in \mathcal{Q}$
- $\text{READY}(a, \text{obs})$ is a pending observation to be delivered to $a \in \alpha$ where $\text{obs} \in \mathcal{L}$ is a logical-space model resulting from an observation
- $\text{CTR}(a, R \Rightarrow R')$ is a control request from $a \in \alpha$ specified by the spatial reaction rule $R \Rightarrow R' \in \mathcal{L} \times \mathcal{L}$

The logical-space computing semantics is modeled abstractly in order to fit different logical and physical models. The semantics is written in an operational style, largely influenced by [2, 78, 1]. It is formalized as a transition system over the space of spatial-computing configurations, specified by seven inference rules.

**Computation**

The rule denoted as $\langle \text{fun} : a \rangle$ models an internal computation performed by agent $a$, i.e., the change of the local state of $a$ specified by the semantics of a given programming language.

$$\langle \text{fun} : a \rangle
\quad E \vdash f;b \rightarrow \lambda E' \vdash b
\quad \frac{}{\langle \alpha, [E \vdash f;b]_a | S | \eta \rangle \rightarrow \langle \alpha, [E' \vdash b]_a | S | \eta \rangle} \quad (5.2)$$

**Observation**

There are three rules for modelling observations.

$$\langle \text{req}_\text{obs} : a, \text{observe}(q) \rangle
\quad \frac{}{\langle \alpha, [E \vdash \text{observe}(q); b]_a | S | \eta \rangle \rightarrow \langle \alpha, [E \vdash b]_a | S | \eta, \text{OBS}(a, q) \rangle} \quad (5.3)$$

$$\langle \text{sense} : \text{OBS}(a, q) \rangle
\quad \frac{(L_q, P_q, \beta_q) = \llbracket q \rrbracket(L, P, \beta)}{\langle \alpha \mid (L, P, \beta) \mid \eta, \text{OBS}(a, q) \rangle \rightarrow \langle \alpha \mid (L', P, \beta') \mid \eta, \text{READY}(a, L_q) \rangle}
\quad \text{where } \beta'(l) = \begin{cases} \beta_q(l) & l \in \text{dom}(L_q) \\ \beta(l) & l \in \text{dom}(L) \land l \not\in \text{dom}(L_q) \end{cases} \quad (5.5)$$
CHAPTER 5. LOGICAL-SPACE COMPUTING

\[
\langle \text{rcv}\_\text{obs} : a, \text{react}(x) \rangle \\
E' = E[x/\text{obs}] \\
\langle \alpha, [E \vdash \text{react}(x); b]_a \mid S \mid \eta, \text{READY}(a, \text{obs}) \rangle \rightarrow \langle \alpha, [E' \vdash b]_a \mid S \mid \eta \rangle 
\]

(5.6)

Rule \langle \text{req}\_\text{obs} : a, \text{observe}(q) \rangle models an agent \(a\) requesting an observation defined by query \(q\) of the logic-space model. The request is defined as \(\text{OBS}(a,q)\) and it is stored in the set of pending requests \(\eta\). Rule \langle \text{sense} : \text{OBS}(a,q) \rangle models the runtime system taking an observation request \(\text{OBS}(a,q)\) from the set of pending requests, interpreting the query over the physical structure, and generating a new logical abstraction \(L_q\).

**Definition 5.10** (Query interpretation). The function \([\cdot] : \mathcal{Q} \times \mathcal{S} \rightarrow \mathcal{S}\) takes a query, a spatial structure, and returns a spatial structure. We denote \((L_q, P_q, \beta_q) = [q](L, P, \beta)\) for a query \(q\) interpreted over the structure \((L, P, \beta)\). \(L_q\) is the logical model resulting from interpreting the query \(q\) over \(L\) and \(P\), and \(\beta_q : \text{dom}(L_q) \hookrightarrow \text{dom}(P_q)\) is the physical interpretation of \(L_q\) over \(P_q\). For the sake of a more readable notation we overload the notation of \((L_q, \theta_P, \theta_\beta) = [q](L, \theta_P, \theta_\beta)\) to \(L_q = [q](L)\), where \(\theta_P\) denotes the empty physical-space model, while \(\theta_\beta\) denotes the empty \(\beta\) function. In addition, we require that:

- \((L_q, P_q, \beta_q)\) is consistent,
- \(P_q \subseteq_P P\), i.e., querying does not introduce new physical locations, and
- \(\forall L_q. \exists q. L_q = [q](L_q)\), i.e., in the absence of a physical model and for any logical model, there exists a query that returns the overall logical model

\((L_q, P_q, \beta_q) = [q](L, P, \beta)\) provides the logical abstraction \(L_q\) interpreting \(q\) over \(L\) and \(P\). The query returns a consistent structure \((L_q, P_q, \beta_q)\). Since \((L, P, \beta)\) can be inconsistent, the query interpretation returns a new physical interpretation \(\beta_q : \text{dom}(L_q) \hookrightarrow \text{dom}(P_q)\) that is required to make \(L_q\) and \(P_q\) consistent. A query interpretation does not introduce new physical locations. Hence, \(P_q \subseteq_P P\). At last, we require that, in the absence of a physical model, for each logical model, there exists a query that returns the overall model.

The physical interpretation \(\beta\) is updated to \(\beta'\) in order to include the new physical interpretations \(\beta_q\). For example, consider that the logical location \text{oilSpill10} is known to be at searchArea0 but at an unknown physical location. After performing a query, the query interpreter finds a physical interpretation for \text{oilSpill10} in \(P\), e.g., an UAV was able to physically detect and geo-reference the oil-spill and provided that information in \(P\).

Note that the rule \langle \text{sense} : \text{OBS}(a,q) \rangle requests that \(L_q \subseteq_L L'\). In other words, the logical-space model \(L\) must be updated to \(L'\) in order to contain the freshly observed logical-space model \(L_q\).

The result is stored in the set of pending requests as \(\text{READY}(a, L_q)\). This rule is responsible for the keeping the logical model and the physical model locally consistent with respect to the observed space, i.e., if two observed physical locations are related, then their logical counterparts are also related. Local consistency is ensured by changing \(L\) to \(L'\) and \(\beta\) to
\(\beta'.\) Rule \(\{rcv\_obs : a, react(x)\}\) delivers an observation \(\text{READY}(a, L_q)\) to \(a\) by assigning \(L_q\) to the local variable \(x\). Note that observation is asynchronous, i.e., an agent first requests an observation, the runtime system gets the necessary data from sensors, and delivers the result as soon as possible.

Control

There are two rules for modelling control actions from spatial agents and one from environmental sources.

\[
\{\text{req} : a, \text{control}(R \Rightarrow R')\} \quad \frac{\langle \alpha, [E \vdash \text{control}(R \Rightarrow R'); b]_a | S | \eta \rangle \rightarrow \langle \alpha, [E \vdash b]_a | S | \eta, \text{CTR}(a, R \Rightarrow R') \rangle}{(5.7)}
\]

\[
\{\text{actuate} : \text{CTR}(a, R \Rightarrow R')\} \quad \frac{R \subseteq L \quad R' \subseteq L' \quad P_R \subseteq P \quad P' \subseteq P'}{\langle \alpha | (L, P, \beta) | \eta, \text{CTR}(a, R \Rightarrow R') \rangle \rightarrow \langle \alpha | (L', P', \beta) | \eta \rangle}{(5.8)}
\]

where \(P_R\) is the canonical physical interpretation of \(R\) given by Definition 5.11.

**Definition 5.11** (Canonical Physical Interpretation). Let \((L, P, \beta)\) be a spatial structure. Let \(R\) be a logical model such that \(R \subseteq L\). \(P_R\) is the canonical physical interpretation of \(R\) iff:

1. \(\text{dom}(P_R) = \{\beta(l) \in \text{dom}(P) | l \in \text{dom}(R), \beta(l)\}\) is defined\}

2. For each \(\text{rel}_i^R\) defined in \(\sigma_R\),

\[
\forall(l_0, \ldots, l_{n-1}) \in \text{rel}_i^R, p_0 = \beta(l_0) \land \ldots \land p_{n-1} = \beta(l_{n-1}) \iff (p_0, \ldots, p_{n-1}) \in \text{rel}_i^{P_R}
\]

In other words, \(P_R\) is the physical model that includes the physical interpretations of locations from \(\text{dom}(R)\). \(P_R\) preserves the relations for locations with physical interpretation. As such, \(R\) and \(P_R\) are consistent. Note that \(R\) is allowed to have locations without physical interpretation without affecting the consistency of \(R\) and \(P_R\). The model \(P_{R'}\) is obtained in the same way.

Note that the semantics allows for requesting partially physically interpreted reactive rules. In fact, a rule is allowed to be completely logical, i.e., where all the locations in the redex are not physically interpreted. This provides means for defining logical locations that acquire physical interpretation later in the execution, or for defining pure logical locations.

Rule \(\{\text{req} \_ \text{ctr} : a, \text{control}(R \Rightarrow R')\}\) models an agent \(a\) requesting a control action over the the logical structure specified by the reaction rule \(R \Rightarrow R'\), where \(R\) specifies the
part of the logic model to be changed and $R'$ specifies how it is intended to be changed. The rule generates a request $CTR(a, R \Rightarrow R')$ in the set of pending requests. Rule $\langle \text{actuate :}CTR(a, R \Rightarrow R') \rangle$ models the runtime system taking a request $CTR(a, R \Rightarrow R')$, checking if it can be applied over the logical and physical space models, and executing the rule over both models. The physical models $P_R$ and $P_{R'}$ are the physical counterparts of the logical models $R$ and $R'$. The rule requests the spatial structure to be locally consistent with respect to $R$ and keeps the structure locally consistent with respect to $R'$. This is formalized later in Section 5.6.

Note that if a single agent observes first the space that is willing to control, locally consistency is ensured and control action can be successfully executed. Nonetheless, in the presence of concurrency, one must ensure that the space being controlled is by agents is free of race-conditions. This is discussed on Section 5.6.

Environment

The effects of the environment are modelled by rule $\langle \text{env :}P' \rangle$, which changes the physical model to $P'$. This rule models effects of the environment over the physical model. For example, a new polygon is generated in $P'$ to model an oil-spill being detected by a sensor on board of an UAV.

$$\langle \text{env :}P' \rangle$$

$$\langle \alpha \mid (L, P, \beta) \mid \eta \rangle \rightarrow \langle \alpha \mid (L, P', \beta) \mid \eta \rangle$$

(5.10)

Example 5.7. Figure 5.8 shows an example of a bigActor as a spatial agent that operates over a bigraphical model of the world. The physical world is modeled as polygons defined using GPS coordinates. The bigActor specified in Figure 5.8 uses the query language defined in Chapter 2 to observe the logical-space for oil-spills and BRRs to move the UAV from its current location to a new one. In order to understand the effects of each rule of the logical-space computing semantics we present the overall execution semantics of the logical-space program of Figure 5.8. The sequence of configurations is given at Figure 5.9. For the sake of a less cluttered execution we do not represent the changes in the state and behavior of $a$. The analysis of the execution trace shows the asynchronous nature of both observation and control. Note that after the step $\langle \text{env :}P'' \rangle$, the oil-spill is modelled at $P''$ but not at $L'$. However, the step $\langle \text{sense :}OBS(a, q) \rangle$ changes $L'$ to $L''$ by adding the logical location $\text{oilSpill10}$ which is physically interpreted by the new polygon in $P''$.

5.6 Correctness of control actions

The execution of a logical-space program is said to be correct if control actions requested at a logical level produce expected effects at the physical level. It is of the programmer
CHAPTER 5. LOGICAL-SPACE COMPUTING

Figure 5.8: Spatial agent modelled as a bigActor.

Figure 5.9: Logical-space execution for the spatial agent of Figure 5.8.
responsibility to write programs that are logically executable, i.e., to request control actions that are feasible over the current logical model. Nonetheless, if a control action is requested over an inconsistent structure, there are no means for ensuring that it will execute correctly, i.e., the control action can be physically non-executable. The runtime system must provide the programmer with means for ensuring that if a control action is logically executable then it is also physically executable.

**Definition 5.12** (Logically executable). Consider the configuration \( \langle \alpha, (L, P, \beta), \eta \rangle \). A control request \( \text{CTR}(a, R \Rightarrow R') \in \eta \) where \( a \in \alpha \) is logically executable if \( R \subseteq L \).

Logically non-executable control requests result from badly written spatial agents, e.g., a requesting to move a vehicle to a location that does not exists. This is not desirable but semantically acceptable. We do not wish to constraint the expressiveness of the language to cope with these kind of mistakes.

**Definition 5.13** (Physically executable). Consider the configuration \( \langle \alpha, (L, P, \beta), \eta \rangle \). A control request \( \text{CTR}(a, R \Rightarrow R') \in \eta \) where \( a \in \alpha \) is physically executable if \( P_R \subseteq P \), i.e., if the canonical physical interpretation of \( R \) is contained in the physical model \( P \).

Physically non-executable control requests result from inconsistencies in the spatial structure. This may occur because a spatial agent requested a control action that is logically non-executable or because the structure became inconsistent due to an environmental action.

**Conditions for physical executable control actions**

In this section we analyse which conditions makes control actions physical executable.

First, we prove the following lemma that relates the logical containment \( \subseteq_L \) with the physical containment \( \subseteq_P \).

**Lemma 5.1.** Let \( (L, P, \beta) \) be a consistent structure. If \( R \subseteq_L L \), then \( P_R \subseteq_P P \) where \( P_R \) is the canonical physical interpretation of \( R \) using \( \beta \).

Proof. Since \( R \subseteq_L L \), by Definition 5.5, \( \text{dom}(R) \subseteq \text{dom}(L) \). Let \( P_R \) be the canonical interpretation of \( R \) over \( P \). By Definition 5.11(1.), \( \text{dom}(P_R) = \{ \beta(l) \in \text{dom}(P) \mid l \in \text{dom}(R), \beta(l) \text{ is defined} \} \). Thus, \( \text{dom}(P_R) \subseteq \text{dom}(P) \).

For each \( \text{rel}_i^{P_R} \), let \( (p_0, \ldots, p_{n-1}) \in \text{rel}_i^{P_R} \). By Definition 5.11(2.), \( \exists (l_0, \ldots, l_{n-1}) \in \text{rel}_i^L \) where \( p_i = \beta(l_i) \). Since \( R \subseteq_L L \), by Definition 5.5, \( (l_0, \ldots, l_{n-1}) \in \text{rel}_i^L \). Since \( (L, P, \beta) \) is consistent, by Definition 5.4, \( (p_0, \ldots, p_{n-1}) \in \text{rel}_i^P \). Thus, by Definition 5.6, \( P_R \subseteq_P P \). \( \square \)

**Theorem 5.1.** Let \( c = \langle \alpha \mid S \mid \eta \rangle \) be a spatial computing configuration where \( S = (L, P, \beta) \), and \( \text{CTR}(a, R \Rightarrow R') \in \eta \) be a control request. If \( S \) is consistent and \( \text{CTR}(a, R \Rightarrow R') \) is logically executable, then \( \text{CTR}(a, R \Rightarrow R') \) is physically executable.
CHAPTER 5. LOGICAL-SPACE COMPUTING

98

Proof. Since $\text{CTR}(a, R \Rightarrow R')$ is logically executable, by Definition 5.12, $R \subseteq_{L} L$. In addition, since $S$ is consistent, by Lemma 5.1, $P_{R} \subseteq_{P} P$. Thus, by Definition 5.13, $\text{CTR}(a, R \Rightarrow R')$ is physically executable. \hfill \Box

Consistency is a sufficient condition for a control action to be physically executable but it is not necessary. This can be demonstrated with a counterexample. Let $c = \langle \alpha \mid (L, P, \beta) \mid \eta \rangle$ be a configuration and $\text{CTR}(a, R \Rightarrow R') \in \eta$ be a control action. Let $R \subseteq_{L} L$ and $P_{R} \subseteq_{P} P$, i.e., $\text{CTR}(a, R \Rightarrow R')$ is logically and physically executable. Assume that there exists a logical model $C$ where $C \subseteq_{L} L$ but $R \not\subseteq_{L} C$. Assume that $(C, P, \beta)$ is inconsistent and thus, $(L, P, \beta)$ is also inconsistent. Let $P_{C}$ be the physical interpretation of $C$. Assume that $P_{C} \subseteq_{P} P$ but $P_{R} \not\subseteq_{P} P_{C}$. Thus, $P_{R} \subseteq_{P} P$ still holds and $\text{CTR}(a, R \Rightarrow R')$ is still physically executable over the inconsistent structure $(L, P, \beta)$.

The previous counterexample shows that consistency is not required as soon as the part of the logical model is not inconsistent. In other words, it is sufficient for the structure to be local consistency with respect to the redex of the control action in order to make it physically executable.

Corollary 5.1. Let $c = \langle \alpha \mid S \mid \eta \rangle$ be a spatial computing configuration where $S = (L, P, \beta)$, and $\text{CTR}(a, R \Rightarrow R') \in \eta$ be a control request. If $S$ is locally consistent with respect to $R$ and $\text{CTR}(a, R \Rightarrow R')$ is logically executable, then $\text{CTR}(a, R \Rightarrow R')$ is physically executable.

Proof. Since $\text{CTR}(a, R \Rightarrow R')$ is logically executable, by Definition 5.12 $R \subseteq_{L} L$. Since $S$ is locally consistent with respect to $R$ and $R \subseteq_{L} L$, by Definition 5.7, $(R, P, \beta)$ is consistent. Since $R \subseteq_{L} L$ and $(R, P, \beta)$ is consistent, by Lemma 5.1, $P_{R} \subseteq_{P} P$. Thus, by Definition 5.13, $\text{CTR}(a, R \Rightarrow R')$ is physically executable. \hfill \Box

In general, local consistency is also not a necessary condition for a logically executable control action to be physically executable. Let $c = \langle \alpha \mid (L, P, \beta) \mid \eta \rangle$ be a configuration and $\text{CTR}(a, R \Rightarrow R') \in \eta$ be a control action where $R \subseteq_{L} L$. Consider that all logical locations specified by $R$ do not have a physical interpretation. Thus, $\text{dom}(P_{R}) = \emptyset$ and, thus, $P_{R} \subseteq_{P} P$ even if the structure is locally inconsistent with respect to $R$. In other words, if a spatial agent requests a purely logical control action, then consistency is not necessary.

Nonetheless, under the assumption that at least one logical location from $R$ has a physical interpretation, then local consistency is a necessary condition for a logically executable control action to be physically executable.

Theorem 5.2. Let $c = \langle \alpha \mid S \mid \eta \rangle$ be a spatial computing configuration where $S = (L, P, \beta)$ and let $\text{CTR}(a, R \Rightarrow R') \in \eta$ be a logically executable control request. Assume that there exists a $l \in \text{dom}(L)$ such that $\beta(l)$ is defined. If $\text{CTR}(a, R \Rightarrow R')$ is physically executable, then $S$ is locally consistent with respect to $R$.

Proof. Since, by hypothesis, $\text{CTR}(a, R \Rightarrow R')$ is logically and physically executable, then by Definition 5.12 and Definition 5.13, $R \subseteq_{L} L$ and $P_{R} \subseteq_{P} P$. Consider the structure $(R, P_{R}, \beta)$. Since, by hypothesis, there exists a $l \in \text{dom}(L)$ such that $\beta(l)$ is defined, the
set \( \text{dom}(P_R) = \{ \beta(l) \mid l \in \text{dom}(R), \beta(l) \text{ is defined} \} \) in Definition 5.11 is non-empty. By the construction of Definition 5.11, if \( P_R \) is the canonical physical interpretation of \( R \) then \( (R, P_R, \beta) \) is consistent. Since \( P_R \subseteq P \), then \( (R, P, \beta) \) is also consistent. Since \( R \subseteq L \), then by Definition 5.7, \( (L, P, \beta) \) is locally consistent with respect to \( R \).

\section{5.7 Using the semantics to enforce local consistency}

In the previous section, we define correctness of control actions in terms of their ability to logically and physically execute. We have also shown that local consistency is a key property to make control actions physically executable. In this section we analyse how one can use the logical-space computing semantics in order to achieve physically executable control actions. Whenever a observation is performed, the semantics rule \text{sense} forces the structure to be local consistent with respect to the observed portion of the logical model.

\begin{lemma}
Consider a transition
\[
\langle \alpha \mid (L, P, \beta) \mid \eta, \text{OBS}(a, q) \rangle \xrightarrow{\text{sense : OBS}(a, q)} \langle \alpha \mid (L', P, \beta') \mid \eta \rangle.
\]
Then \( (L', P, \beta') \) is locally consistent with respect to \( L_q \) where \( (L_q, P_q, \beta_q) = [q](L, P, \beta) \).
\end{lemma}

\begin{proof}
By Definition 5.10, \( (L_q, P_q, \beta_q) \) is consistent. By Definition 5.10, \( P_q \subseteq P \). Thus, by Definition 5.4, \( (L_q, P_q, \beta_q) \) is consistent. By definition of \( \beta' \) in Equation 5.5, \( \beta' \) restricted to \( L_q \) is equal to \( \beta_q \). Thus, by Definition 5.4, \( (L_q, P, \beta') \) is consistent. By the premisses of the rule \( \langle \text{sense} : \text{OBS}(a, q) \rangle \), \( L_q \subseteq L' \). Thus, by Definition 5.7, \( (L', P, \beta') \) is locally consistent with respect to \( L_q \).
\end{proof}

On the other hand, a control request can only be executed if the structure is locally consistent with respect to the redex that specifies that portion of the logical model to be modified.

\begin{lemma}
Consider the transition
\[
\langle \alpha \mid (L, P, \beta) \mid \eta, \text{CTR}(a, R \Rightarrow R') \rangle \xrightarrow{\text{actuate : CTR}(a, R \Rightarrow R')} \langle \alpha \mid (L', P', \beta) \mid \eta \rangle.
\]
\( (L, P, \beta) \) is locally consistent with respect to \( R \) and \( (L', P', \beta) \) is locally consistent with respect to \( R' \).
\end{lemma}

\begin{proof}
By the premisses of rule \( \langle \text{actuate} : \text{CTR}(a, R \Rightarrow R') \rangle \) (Equation 5.8), \( R \subseteq L \) and \( P_R \subseteq P \). By the construction of Definition 5.11, \( (R, P_R, \beta) \) is consistent. Since \( R \subseteq L \) and \( P_R \subseteq P \), then by Definition 5.7, \( (L, P, \beta) \) is locally consistent with respect to \( R \).

By the premisses of rule \( \langle \text{actuate} : \text{CTR}(a, R \Rightarrow R') \rangle \) (Equation 5.8), \( R' \subseteq L' \) and \( P_{R'} \subseteq P' \). By Definition 5.11, \( (R', P_{R'}, \beta) \) is consistent. Since \( R' \subseteq L' \) and \( P_{R'} \subseteq P' \), then by Definition 5.7, \( (L', P', \beta) \) is locally consistent with respect to \( R' \).
\end{proof}
Consider a configuration \( c_0 = \langle \alpha, a \mid S \mid \eta \rangle \) where \( S_0 = \langle L_0, P_0, \beta_0 \rangle \) and
\[
[E \vdash \text{observe}(q); \text{ready}(x); \text{control}(x \Rightarrow f(x))]_a
\]
for some \( f : \mathcal{L} \rightarrow \mathcal{L} \) and some query \( q \). If the spatial agent \( a \) requests a control action \( \text{CTR}(a, x \Rightarrow f(x)) \) where \( \text{dom}(x) \neq \emptyset \) then there exists a trajectory starting from the configuration \( c_0 \) where \( \text{CTR}(a, x \Rightarrow f(x)) \) is requested and actuated.

**Proof.** We prove by constructing \( t = c_0 \xrightarrow{\lambda_0} c_1 \xrightarrow{\lambda_1} c_2 \xrightarrow{\lambda_2} c_3 \xrightarrow{\lambda_3} c_4 \xrightarrow{\lambda_4} c_5 \), where:

- \( c_0 = \langle \alpha, [E \vdash \text{observe}(q); \text{ready}(x); \text{control}(x \Rightarrow f(x))]_a \mid S \mid \eta \rangle \)
- \( \lambda_0 = (\text{req}_\text{obs} : a, \text{observe}(q)) \)
  - By definition of the rule denoted as \( (\text{req}_\text{obs} : a, \text{observe}(q)) \) (Equation 5.3), the spatial agent reduces to \( [E \vdash \text{ready}(x); \text{control}(x \Rightarrow f(x))]_a \) and an observation request \( \text{OBS}(a, q) \) is generated in \( \eta_1 \). \( \lambda_0 \) does not affect the structure. Thus, \( S_1 = S_0 \).
- \( c_1 = \langle \alpha, [E \vdash \text{ready}(x); \text{control}(x \Rightarrow f(x))]_a \mid S_1 \mid \eta, \text{OBS}(a, q) \rangle \)
- \( \lambda_1 = (\text{sense} : \text{OBS}(a, q)) \)
  - By Lemma 5.2, the rule \( (\text{sense} : \text{OBS}(a, q)) \) can fire, making \( S_2 \) is locally consistent with respect to \( L_q \), where \( (L_q, P_q, \beta_q) = \llbracket q \rrbracket (L_1, P_1, \beta_1) \).
- \( c_2 = \langle \alpha, [E \vdash \text{ready}(x); \text{control}(x \Rightarrow f(x))]_a \mid S_2 \mid \eta, \text{READY}(a, L_q) \rangle \)
- \( \lambda_2 = (\text{rcv}_\text{obs} : a, \text{ready}(x)) \)
  - By definition of the rule \( (\text{rcv}_\text{obs} : a, \text{ready}(x)) \) (Equation 5.6), \( \text{READY}(a, L_q) \) is consumed, and \( [E' \vdash \text{control}(x \Rightarrow f(x))]_a \) where \( E' = E[x/L_q] \).
- \( c_3 = \langle \alpha, [E' \vdash \text{control}(x \Rightarrow f(x))]_a \mid S_3 \mid \eta \rangle \)
- \( \lambda_3 = (\text{req}_\text{ctr} : a, \text{control}(R \Rightarrow R')) \)
  - By definition of the rule denoted as \( (\text{req}_\text{ctr} : a, \text{control}(x \Rightarrow f(x))) \) (Equation 5.7), the spatial agent reduces to \( [E' \vdash]_a \) and a control request \( \text{CTR}(a, x \Rightarrow f(x)) \) is generated in \( \eta_4 \). \( \lambda_3 \) does not affect the structure so local consistency with respect to \( L_q \) is preserved in \( S_4 \).
- \( c_4 = \langle \alpha, [E' \vdash]_a \mid S_4 \mid \eta, \text{CTR}(a, x \Rightarrow f(x)) \rangle \)
• $\lambda_4 = \langle \text{actuate : } \text{CTR}(a, x \Rightarrow f(x)) \rangle$

  - Since $S_4$ is locally consistent with respect to $x = L_q$. Thus, the rule $\langle \text{actuate : } \text{CTR}(a, x \Rightarrow f(x)) \rangle$ (Equation 5.8) can fire. By Lemma 5.3, $\text{CTR}(a, x \Rightarrow f(x))$ is actuated, i.e., it is executed and removed from the bag of requests. By the same lemma, $S_5$ is now locally consistent with respect to $f(x)$

• $c_5 = \langle \alpha, [E' \vdash \cdot]_a \mid S_5 \mid \eta \rangle$

Theorem 3.2 shows that if a spatial agent observes the space that is willing to control than there is an execution that actuates its control request. In an analogy to control theory, we call these agents, feedback.

**Example 5.8.** Figure 5.10 provides an example of a feedback controller. Note that the

```
BigActor hosted_at uav0 with_behavior{
  loop{
    observe(CHILDREN(PARENT(HOST)))
    react{
      case obs.contains(oilSpill0) => control(MOVE_HOST_TO(oilSpill0))
      case _ =>
    }
  }
}
```

Figure 5.10: Example of a feedback specified as a bigActor.

bigActor executes MOVE_HOST_TO(oilSpill0) only if it observes oilSpill0.

Being a feedback agent is not sufficient to make control requests executable. This is due to two factors: environmental actions and concurrency. The environment or other agents may preclude execution by changing the physical structure after the control request is made by the feedback agent.

Since space is a first class concept in our model, it can express the partitioning of agents in space to avoid conflicts due to concurrency in time. This is formalized in Theorem 5.4. To set up the theorem, we define an unobtrusive environment (Definition 5.14) and then a partitioned control policy for a set of concurrent agents (Definition 5.15).

**Definition 5.14** (Unobtrusive environment). Let $c = \langle \alpha \mid (L, P, \beta) \mid \eta, \text{CTR}(a, R \Rightarrow R') \rangle$ be a spatial structure. An environmental action $\langle \text{env : } P' \rangle$ is unobtrusive with respect to $\text{CTR}(a, R \Rightarrow R')$ if $P_R \subseteq_P P'$ where $P_R$ is the canonical interpretation of $R$. 
CHAPTER 5. LOGICAL-SPACE COMPUTING

Definition 5.15 (Partitioning control). A set of agents \( \{a_0, a_1, \ldots, a_{n-1}\} \) follow a partitioning control if there exists logical models \( L_0, L_1, \ldots, L_{a_{n-1}} \) such that, for any two spatial agents \( a_i \) and \( a_j \), \( \text{dom}(L_{a_i}) \land \text{dom}(L_{a_j}) = \emptyset \), and for any control request \( \text{CTR}(a_i, R_i \Rightarrow R'_i) \), \( \text{dom}(R_i) \subseteq \text{dom}(L_{a_i}) \) and \( \text{dom}(R'_i) \subseteq \text{dom}(L_{a_i}) \).

If each agent operates only in its own space, local consistency can be preserved.

Theorem 5.4. Consider a configuration \( c_0 = \langle a_0, a_1, \ldots, a_{m-1}, a_w \mid S_0 \mid \eta_0 \rangle \) where \( S_0 = (L_0, P_0, \beta_0) \) is consistent. Let \( [E \vdash \text{control}(R \Rightarrow R')]_{a_w} \) be a spatial agent where \( R \subseteq L_0 \). Assume that \( a_0, a_1, \ldots, a_{m-1} \) follow a partitioning control policy. Assume that all environmental actions are unobtrusive with respect to \( R \). Let \( t = c_0 \xrightarrow{\lambda_0} c_1 \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_{n-1}} c_n \) be a trajectory where there exists a \( \lambda_i = \langle \text{req_ctr} : a_w, \text{control}(R \Rightarrow R') \rangle \). \( \text{CTR}(a, R \Rightarrow R') \) in any configuration following \( c_{i+1} \), i.e., in any \( c_n \) with \( n > i + 1 \) can be actuated.

Proof. We prove by showing that local consistency with respect to \( R \) is preserved. By hypothesis, \( S_0 \) is consistent. By hypothesis, \( R \subseteq L_0 \). Thus, by Definition 5.7, \( S_0 \) is locally consistent with respect to \( R \).

There are two kinds of semantic rules that change the physical space making space locally inconsistent with respect to \( R \): \text{actuate} and \( \langle \text{env} : P' \rangle \). Let \( \lambda_j \) be an arbitrary event where \( j > 0 \) and \( j \neq i \). Let \( \lambda_j = \langle \text{env} : P' \rangle \) By assumption, all environmental actions are unobtrusive with respect to \( R \). By Definition 5.14, \( P_R \subseteq P_{j+1} \). Thus, \( S_{j+1} \) remains locally consistent with respect to \( R \).

Let \( \lambda_j = \langle \text{actuate} : \text{CTR}(a_k, R_k \Rightarrow R'_k) \rangle \). By construction of Definition 5.15, for all agents \( a_k \neq a_w \), \( \text{CTR}(a_k, R_k \Rightarrow R'_k) \), \( \text{dom}(R_k) \land \text{dom}(L_{a_w}) = \emptyset \) and \( \text{dom}(R'_k) \land \text{dom}(L_{a_w}) = \emptyset \). Since, \( \text{dom}(R) \subseteq \text{dom}(L_{a_w}) \), \( S_{j+1} \) remains locally consistent with respect to \( R \).

By hypothesis, \( \lambda_i = \langle \text{req_ctr} : a_w, \text{control}(R \Rightarrow R') \rangle \). Thus, \( c_{i+1} = \langle a_0, a_1, \ldots, a_{m-1}, a_w \mid S_{i+1} \mid \eta_i, \text{CTR}(a_w, R \Rightarrow R') \rangle \). Without lost of generality, assume that \( n = j + 1 \). Since, local consistency with respect to \( R \) is preserved for any arbitrary \( S_{j+1} \), by Lemma 5.3, \( \lambda_n = \langle \text{actuate} : \text{CTR}(a_i, R \Rightarrow R') \rangle \) can semantically execute producing \( c_{n+1} \).

Theorem 5.4 shows that under controlled concurrency and environment, one can write spatial agents that eventually have their control requests actuated.

5.8 Final remarks

In this chapter, we introduce logical-space computing - a new spatial computing paradigm where programs interact with a logical abstraction of spatial location of computing machines while a runtime system is in charge of mediating the interactions between the logical space and the physical space. Logical-space computing handles a hybrid model of space since it combines both physical and symbolical abstractions of space. It contrasts with other hybrid approaches in the literature since it keeps the programs free of any physical space information. By constraining programs to operating over symbolic locations, logical-space computing removes the burden of correctly define physical behaviors from the programmer.
We provide a definition for logical and physical spaces and a physical interpretation that binds these two abstractions together. The physical interpretation is modeled as a partial function, i.e., we allow logical locations to not have a physical interpretation. This design decision has two practical implications. One can define logical locations that are “purely” logical, e.g., modeling the logical location of a cloud-based application on a smartphone, while its true physical location is unknown and not relevant. One can also define logical locations that do not have a physical interpretation at a given point in time but acquire a physical interpretation later in the execution, e.g., a person with a smartphone is known to be inside a car although, the GPS signal in the smartphone is too weak to provide a physical location. We say that the structure that entails a logical model and a physical model is consistent if the physical interpretation preserves the relations from the logical model in the physical model.

Logical-space programs are specified as spatial agents. A spatial agent have local state, local behavior, and can interact with the logical space by asynchronously requesting observations and control actions. We provide the formal semantics for spatial agents in an operational style as a set of inference rules that define a transition system over the universe of spatial-computing configurations.

We analyze the correctness of logical-space executions in terms of the executability of the requested control actions. Consistency is a key property to ensure that control actions are executable. Theorem 5.1 shows that consistency is a sufficient condition for logically executable control actions to be physically executable. As a corollary of Theorem 5.1, we prove that it is sufficient for the structure to be locally consistent for the control action to be physically executable. In fact, if the control action is not purely logical, i.e., there exists at least a node in the redex that has a physical interpretation, then local consistency is a necessary condition. This is formalized by Theorem 5.2.

We explore how to use the logical-space computing semantics to enforce local consistency and, thus, to ensure that control actions are physically executable. We propose a programming pattern for spatial agents called feedback. Feedback spatial agents get their requests executed by observing the space that are willing to control. This is formalized by Theorem 5.3.

Nonetheless, being a feedback is not sufficient for getting control actions executed. The environment and concurrency can make control actions non-executable. We introduced a condition over the environment named unobtrusive that requests that requests the environment to keep control requests physically executable. We also introduced a concurrency condition named partitioning control that requests spatial agents to actuate over disjoint spaces. Theorem 5.4 shows that, under controlled concurrency and environmental actions, logically executable control actions operating over consistent structures are eventually actuated.
Chapter 6

Case study: oil-spill monitoring mission

We have used Logical Space Programming to execute an oil spill monitoring mission in the Atlantic Ocean as a case study with a UAV, a satellite, drifters, and three control stations distributed over 20 km. The case study is a collaborative effort with the European Maritime Safety Agency (EMSA), the Portuguese Air Force Pitvant project, and the Portuguese Navy. Our contribution is the logical-space program, and the runtime system coordinating all the entities. Installation of the runtime system on the Air Force and Naval hardware is a joint effort. The mission itself is executed by the Air Force, Navy, and EMSA. The author of this dissertation participated in the 30 person execution team as an officer of the Air Force holding the rank of Captain.

Our field test demonstrates the role of unmanned vehicles and sensors for complementing satellites for collecting evidence of illegal bilge dumping. The software infrastructure implemented for this case study is named Logical-Space Runtime System (LSRS) and follows the semantics described in Chapter 5. LSRS was designed specifically for this field test. It is our first step towards programming mobile computing systems using the logical-space computing.

6.1 The bilge dumping problem

Illegal bilge dumping has been identified as a serious form of maritime pollution. States holding jurisdiction over large waters need technology and policies to reduce bilge dumping. EMSA is an agency created by the European Commission (EC) with the specific mission of providing the European member states with means to enforce maritime safety. Approximately half of EMSA’s budget (around 58.8 million of Euros in 2013 [79]) is dedicated to the field of marine pollution. Its activity focus on responding to ship-source marine pollution, firstly oil pollution and then pollution by hazardous and noxious substances.

The global magnitude of the problem is difficult to assess due to the lack of historical
data [80]. Nonetheless, there are increasing reports of tankers discharging in open waters without the consent of the country of jurisdiction. According to EMSA, there were 674 confirmed mineral oil-spills out of 2579 inspections in European Member State coastal waters between 2007 and 2010 [81].

Gathering sufficient evidence to prosecute a suspect of illegal bilge dumping is hard. The testimony of an illegal discharge (either in person or video footage) is not sufficient. One must gather physical evidences of the crime in the form of water samples both from the vicinities of the tanker and the tanker itself [82]. The most common technique for detecting oil spill is satellite imagery, from optical and Synthetic Aperture Radar (SAR) sources [83], correlated with Automatic Identification System (AIS) information [84, 85]. AIS is the de facto identification protocol for Vessel Traffic Services (VTS) [86]. Vessels broadcast their GPS location and a unique identification number, known as MMSI1, using VHF radio-frequency transmission.

In order to perform this mission, EMSA provides three tools: CleanSeaNet, SafeSeaNet, and IMDatE. CleanSeaNet is a satellite-based monitoring system for oil-spill detection in European waters [87, 88, 89]. CleanSeaNet is a service based on SAR imagery covering all European sea areas. SafeSeaNet is a vessel traffic monitoring and information system that collects vessel traffic information using AIS. IMDatE stands for Integrated Maritime Data Environment [90] and is used as a tool for integrating maritime data from different sources such as CleanSeaNet and SafeSeaNet. In order to search for illegal bilge dumping, imagery from CleanSeaNet is analysed and correlated with information from SafeSeaNet. This analysis is performed at IMDatE, which can provide both real-time and historical maritime data. If a oil spill is detected, an alert message is delivered to the country of interest. Figure 6.1 illustrates CleanSeaNet and SafeSeaNet providing evidence for bilge dumping off the coast of Spain.

Using satellite imagery and AIS is not sufficient for collecting evidence of illegal bilge dumping. Tasking satellites is limited in time and space and is influenced by the presence of clouds [83]. Moreover, there is the need for collecting water samples from the vicinity of the oil-spill and from suspect tankers. Thus, remote sensing is not sufficient. Academia and law enforcement organizations have been interested in supplementing satellite imagery with data collected from autonomous vehicles, such as Autonomous Underwater Vehicles (AUVs), Unmanned Aerial Vehicles (UAVs), and Autonomous Surface Vehicles (ASVs) [91, 92, 93, 94]. These vehicles have been studied as an alternative and a complement of manned systems to achieve persistent surveillance of maritime environments, including bilge dumping detection and monitoring [95, 92, 96]. The aim is to combine the aerial sensing capabilities of UAVs, satellite imagery, AUVs for sampling the water in the vicinity of a tanker, and ASVs as communication relays [68]. AUVs have limited communication capabilities, often low-rate acoustic modems. The ASVs are used to relay information between AUVs and other players. Our case study explores the use of vehicles and sensors to complement satellite imagery for collecting evidence of illegal bilge dumping. Next we present the specification that influenced

---

1Acronym for Maritime Mobile Service Identity.
CHAPTER 6. CASE STUDY: OIL-SPILL MONITORING MISSION

Figure 6.1: CleanSeaNet SAR image with evidences of bilge dumping acquired in June 2009 off the coast of Spain. The suspected vessel was also detected in the SAR image. The satellite image is compared with SafeSeaNet AIS database in order to identify the suspected ship. Source: EOMag [4].

our design decisions.

Specification

The procedure of collecting evidence of bilge dumping goes as follows [97]. An operator identifies a feature in the sea that resembles an oil-spill using satellite imagery, either SAR or optical. The operator analyses the trajectories of vessels in the vicinities of the oil-spill using AIS historical data. A vessel is labelled as a suspect if it shows a trajectory that matches the oil-spill both in space and time. Further evidence is needed to make a vessel formally a suspect. There is the need for in situ data (imagery and identification) of the vessel and the oil-spill, and a water sample from both the oil-spill, and from the suspect vessel’s tanker.

The specification for the field test was provided by EMSA, the Portuguese Air Force, and the Portuguese Navy. EMSA provided the requirements that drove the design of the case study, while the Portuguese Air Force and the Portuguese Navy provided inputs regarding the
join operation of aerial and surface vehicles. Next we enumerate each requirement together with a brief discussion.

**Requirement 6.1.** _Demonstrate the use of alternative means for collecting evidence of illegal bilge dumping activity._

Tasking satellites is limited in time and space. EMSA’s CleanSeaNet service uses 4 low-earth orbit satellites, which acquire images in segments of up to 1400 km and swaths of up to 500 km [98]. Nonetheless, images must be tasked in advance and take from 30 to 60 minutes to be delivered. Therefore, EMSA, in coordination with the involved Member States, currently uses aerial imagery from conventional aircraft to persistently survey oil-spills and vessels at real-time. EMSA is interested on studying the use of alternative means such as autonomous vehicles for collecting data from vessels and oil-spills that complement the use of satellites [99].

The Portuguese Air Force and the Portuguese Navy decided to address Requirement 6.1 using UAVs for collecting imagery and AIS information, drifters for forecasting the trajectory of the oil-spill and for sampling the oil-spill, and a Navy vessel to deploy the drifters. The operation of such a spatially distributed heterogeneous system posed operational-specific requirements that are discussed next.

**Requirement 6.2.** _Implement a software infrastructure that collects information from heterogeneous sources, e.g., UAV’s autopilot and AIS receivers, and generates a common location model_

Spatial information is provided by different systems in different formats, e.g., telemetry messages sent by the UAV autopilot and AIS messages broadcast by drifters and vessels. There is the need to find a common location model and software means to collect information from heterogeneous sources and generate location abstractions on demand.

**Requirement 6.3.** _Implement networking capabilities to provide consistent location model to all mission players, e.g., UAV’s operators and vessel’s commanders_

The operation of an UAV often requires more than one operator taking control of the UAV from spatially distributed ground control stations during a single mission. For example, it is common to have an operator at an airfield responsible for taking-off and landing, and an operator onboard a vessel to conduct the operation at increased range. Operators must know at all time where the UAV is located and which operator has control authority. All operators need consistent views of the location and connectivity of the UAV location model as well as its connectivity.

**Requirement 6.4.** _Implement a programming model that specifies the mobility of the UAV, abstracting low-level physical details._

Conventional UAVs autopilots provide operators with a GPS waypoint controller interface, i.e., an operator specifies the latitude and longitude of the desired destination and the
autopilot takes care of controlling the actuators. In our field test, the oil-spill is delimited by the drifters that provide its GPS location. In a conventional setting, in order to send the UAV to visit the oil-spill, the UAV operator would have to read the location of the drifters and calculate a GPS coordinates for the waypoint, e.g., at the centroid of the polygon formed by the drifters. This is undesirable since the oil-spill is constantly moving creating room for the operator to make a mistake and send the UAV to a undesirable location.

Our scenario aimed at emulating, as close as possible, a mission of vehicles and sensors collecting evidence of a bilge dumping activity from a tanker crossing waters under Portuguese jurisdiction. The Portuguese Navy emulated the oil-spill by releasing 100kg of popcorn in the ocean in the southern coast of Portugal, around 6 miles off the coast of the village of Portimão. EMSA tasked a satellite to take a high-resolution optical picture of the emulated oil-spill (see Figure 6.2).

![Figure 6.2: High-resolution satellite image from the oil spill. Courtesy of EMSA.](image)

The reminder of this chapter goes as follows. In Section 6.2 we present the vehicles and sensors used in the field test. This addresses Requirement 6.1. In Section 6.3 we describe the software architecture named Logical-Space Runtime System, which provides the means for programming the oil-spill monitoring scenario using logical-space computing as discussed in Chapter 5. This addresses Requirement 6.4.

The requirements 6.2 and 6.3 are addressed in Section 6.4 where we describe the Logical-Space Execution Engine (LSEE). LSEE is responsible for interfacing vehicles and sensors and building location abstractions in the form of bigraphs. Bigraphs are used to model the location and connectivity of the UAV, the Navy vessel, and the drifters, as well as the location of the oil-spill, and other spaces of interest such as the airfield, the search
area, and the segregated airspace. LSEE uses the Robot Operating System (ROS) - an open-source middleware for robotic applications [100] - in order to create a software layer that abstracts the heterogeneous hardware. LSEE fetches physical information from ROS and generates consistent bigraphical abstractions. This addresses Requirement 6.2. LSEE provides a communication infrastructure to share location information between spatially distributed components. LSEE establishes a network between UAV ground control stations and the Navy vessel using the internet and provides means for flooding location information throughout the network. This way, further local processing gives each actor a consistent bigraphical view of the state of the mission. This addresses Requirement 6.3. LSEE also provides means for controlling UAVs logically. A programmer specifies control actions in a logical-space, while LSEE is responsible for interpreting them and generate the corresponding command in the physical space. For example, a programmer uses a single command `MOVE_HOST_TO(oilSpill0)` to send a UAV to the oil-spill location even though the physical interpretation of `oilSpill0` might change over time. This addresses Requirement 6.4.

### 6.2 Vehicles and Sensors

The Requirement 6.1 is addressed using the following machinery: one Air Force UAV, three ground control stations, four drifters that broadcast their position using AIS, and one Navy vessel equipped with a small speedboat. The UAV is used to collect imagery from the oil-spill and AIS information broadcast by vessels and drifters. The UAV is controlled from three different ground control stations. One is solely dedicated for take-off and landing from the Portimão aerodrome. The other ground control stations are used to control the mission. They are placed near the shore and on board the Navy vessel in order to increase the range of the mission. The drifters are used to forecast the trajectory of the oil-spill. They are equipped with GPS and an AIS transmitter to broadcast their location. The Navy vessel are used for flushing the popcorn emulating a tanker flushing its tanks. The speedboat is used to release the drifters in the oil-spill.

We used two kinds of UAVs developed at the Portuguese Air Force Academy under the PITVANT project, the Alfa and the Alfa-Extended (Figure 6.3). The Alfa-Extended is a gas-powered UAV with 3 m wingspan, equipped with a Piccolo autopilot for stable low-level control, and a PC-104 computing board for high-level control and vision processing. Each UAV is equipped with a gimballed optic camera and an AIS receiver. Figure 6.4 shows a picture collected from the Alfa-Extended UAV depicting the oil-spill (yellow patch), the Navy vessel (left-upper corner) and the small speedboat used to deploy the drifters in the middle.

The Unmanned Aerial System included three ground control stations (GCS) denoted as `gcs0`, `gcs1`, and `gcs2`. `gcs0` was situated in an air field and was responsible for take-off and landing maneuvers, `gcs1` was located at the shore and took control authority of the UAVs during emergencies and unforeseen situations, and `gcs2` was located at the shore and was responsible for the UAV mission. In one particular scenario, `gcs2` was located on a Navy
vessel to extend the operational range of the mission.

The drifters used in this demonstration were AIS beacons commonly used for locating fishing nets. They were equipped with GPS and transmitted their position up to a range of 10 miles. Drifters were identified by unique MMSI numbers. Figure 6.5 provides a screenshot of the visualization tool implemented to follow the mission execution. One can see the physical location of the UAV and the drifters.

6.3 Logical-Space Runtime System

Logical-space computing provides the programmer with a logical representation locations of computation. LSRS provides a bridge between the logical-space abstraction and the physical space. LSRS addresses Requirement 6.4 by specifying logically the spatial behavior
of a UAV using bigActors. For more about BigActors and the BigActor Programming Language see Chapter 3 and Chapter 4 of this dissertation.

The logical-space runtime system is implemented as an extension of the the BigActor Runtime System (BARS) presented in Chapter 4. The model checker BigMC which serves as a bigraphical execution engine in BARS is now replaced by the Logical-Space Execution Engine (LSEE). The LSEE interacts with the physical world and produces bigraphical abstractions. The LSEE is implemented on ROS, a popular open-source middleware for robotics. This implementation is developed primarily for specifying the spatial behavior of vehicles and sensors performing an environmental monitoring mission. Its interfaces with physical components are kept application-agnostic in order to facilitate its extension to program other kinds of physical systems.

Figure 6.6 depicts the logical-space runtime system architecture. The left-hand side of Figure 6.6 shows bigActor instances running over the BigActor Runtime System (BARS). BARS is introduced in Chapter 4 and enables programming with bigActors over bigraphical models of space. The right-hand side of Figure 6.6 depicts the LSEE, which provides BARS with a bigraphical abstraction and implements the logical-space computing semantics. It replaces BigMC used as a bigraphical execution engine in the implementation described in Chapter 4 with an execution engine that interfaces with the physical world, i.e., ROS middleware. LSEE is described in Section 6.4.

### 6.4 Logical-Space Execution Engine

LSEE has the following roles: serve as a middleware for sensors and actuators, generate bigraphs from physical properties, consistent distribute bigraphs constructed at one location to the other locations of computation, and interpret BRRs into control commands that can be
executed by the appropriate actuator. Next we present the components that are responsible for these tasks.

**Middleware**

The middleware component of the LSEE is named `ros_vehicle`. It addresses Requirement 6.2 by creating a middleware layer that abstracts hardware such as the UAV autopilot and AIS receivers. `ros_vehicle` instances run on computing entities that we call `vehicles`. Each `ros_vehicle` instance is equipped with software drivers named `plugins` and with additional services for inter-vehicle communication and naming.

Plugins are implemented over ROS, which provides a publish-subscribe communication mechanism. Plugins interact with software and hardware components that produce location and connectivity information or consume control commands. These components can be for example a GPS device, an autopilot, a computer vision system, or a cloud-based location service accessed over the internet.

For the oil-spill scenario we implemented five plugins. The *Autopilot Plugin* handles the execution of GPS waypoints over the autopilot and fetches the UAV state information, like GPS location, velocity, and control authority. The *AIS Plugin* receives, decodes, and filters AIS messages received from an onboard AIS receiver. AIS messages are broadcasted from vessels and drifters, providing GPS location information and the unique identifier of the vessel or drifter, i.e., the MMSI. The *Camera Plugin* uses a video camera driver to capture and process video frames\(^2\).

\(^2\)The purpose of the camera plugin is to automatically process video frames, automatically detect oil-spills, and produce the corresponding location information. At the time of the demonstration, the camera plugin was incomplete. The detection of oil-spills was performed with an operator in the loop.
A plugin has a well-defined interface. It can subscribe *mobility commands* and publish *physical properties* defined using ROS messages. Mobility commands specify actions to be executed by physical platforms, e.g., a GPS waypoint to be executed by an UAV. Figure 6.7 provides the message definition for *MobilityCommand*, where *vehicleID* is the name of the vehicle that is going to execute the command, *timeStamp* is a globally-defined time stamp, *type* is an integer that denotes the kind of command to execute, e.g. *type=1* denotes a waypoint command, and *cmdParam* is a list of parameters defined using JSON that parametrizes the command, e.g., the GPS latitude and longitude of a waypoint.

```plaintext
uint64 timeStamp # Timestamp in milliseconds since Unix Epoch
uint8 vehicleId # Unique id of vehicle to execute the MobilityCommand
string props # Properties as JSON
uint8 type # Type
  uint8 MT_WAYPOINT=0
  uint8 MT_LOITER=1
  uint8 MT_OTHER=2
```

Figure 6.7: ROS message definition for *MobilityCommand*.

Physical properties specify spatial information provided from physical platforms. A physical property may contain static information, such as a polygon describing the airfield area, or dynamic information, such as the GPS location and connectivity of an UAV or the oil-spill location. Figure 6.8 provides the message definition for *PhysicalProp*, where *vehicleId* is the ID of the vehicle that produced the physical property, *timeStamp* is a globally-defined time stamp, *locations* is a list of locations, and *connections* is a list of connections.

```plaintext
uint64 timeStamp # Timestamp in milliseconds since Unix Epoch
uint8 vehicleId # Unique id of vehicle that produced the PhysicalProp
Location[] locations # List of locations
Connection[] connections # List of connections
```

Figure 6.8: ROS message definition for *PhysicalProp*.

Locations are defined by the type *Location* specified by the ROS message in Figure 6.9, where *locationId* is a unique ID, *name* is a unique name, *props* is a list of properties written in JSON that defines the location, e.g., GPS coordinates defining the vertices of a polygon, and *type* is the location type, i.e., a GPS point, a circle, or a polygon.

Connections are defined by the type *Connection* specified by the ROS message in Figure 6.10, where *connectionId* is a unique ID, *name* is a unique name, *locIds* is a list of IDs of the locations that share this connection, and *type* is the connection type, e.g., Wifi, 3G, or a command and control link.

The *ros_vehicle* is also equipped with a *Naming Service* and a *Communication Service*. The Naming Service is responsible for assigning unique names to physical properties and can
be implemented using different naming conventions. The Naming Service implemented for
the oil-spill scenario uses the autopilot serial number to identify the location of the UAV and
the AIS MMSI to identify the locations of the drifters.

The Communication Service specifically addresses Requirement 6.3. It is responsible for
sharing local observations between ros_vehicles. For the oil-spill scenario, the Commu-
nication Service supports the TCP and UDP transport protocols. When operating over a
local WiFi network we use UDP to avoid congestion. When operating over the internet
we use TCP. The communication service assumes that each ros_vehicle has a unique IP.
For our field test, the communication service shares physical properties between different
ros_vehicles at different ground stations. Computers running ros_vehicles had internet
access through 3G USB wireless modems. The internet provider used in the field test pro-
vided a public IP for each 3G modem. This practice can not be used in general since some
internet providers deliver NAT\(^3\) private IPs to these devices. This problem can be overcome
using port forwarding.

\(^3\)Network Access Translation is a methodology of remapping one IP address space into another by
modifying network address information in datagram packet headers while they are in transit across a router.
CHAPTER 6. CASE STUDY: OIL-SPILL MONITORING MISSION

Generation of bigraphs

Recall the logical-space computing formalized in Chapter 5. Lemma 5.2 states that a transition $c \xrightarrow{\text{sense.\text{OB}(a,q)}} c'$ makes the spatial structure at configuration $c'$ locally consistent. In other words, whenever a spatial agent requests an observation, the runtime system is responsible for making the logical-space abstraction locally consistent with respect to the portion of observed physical space. The logical-space runtime system ensures local consistency through the Bigraph Driver. When a bigActor performs a query, the BigActor Scheduler requests the Bigraph Manager for the respective graphical abstraction. The Bigraph Manager gets bigraphs from the Bigraph Driver, which interacts with ros\_vehicle to ensure consistency. Recall the spatial structure $S = (B, P, \beta)$ presented in Section 5.3 where the logical-space $B$ is a bigraph and the physical-space $P$ is a set of polygons and relations over those polygons, namely $\text{parent}_P$ and $\text{linking}_P$. Given any $P$, the Bigraph Driver generates a logical-space model $B$ and $\beta$ such that $S$ is consistent.

The software component that implements the Bigraph Driver subscribes to messages of kind PhysicalProp which model the physical-space and publishes messages of kind Bigraph which model the logical-space. The Bigraph message definition is provided in Figure 6.11, where $\text{bgm}$ is a bigraph term specified using BGM. The syntax for BGM is introduced in Chapter 4.

```
uint64 timeStamp # Timestamp in milliseconds since Unix Epoch
string bgm # Bigraph defined as bgm term
```

Figure 6.11: ROS message definition for Bigraph.

The Bigraph Driver names each bigraph using the name of the corresponding physical interpretation, which is uniquely assigned by the ros\_vehicle Naming Service.

Recall that physical spaces in LSEE are specified using messages of kind PhysicalProp, defined by the ROS message in Figure 6.8. A message PhysicalProp defines a list of locations that can be points, polygons, or circles. Also recall the parenting relation for polygons where $(p',p) \in \text{parent}_P$ iff $p'$ is the smallest polygon or circle that totally contains the the point, polygon or circle defined by $p$. The relation $\text{parent}_P$ is checked using the JTS topology suite library [101].

The linking graph is generated using connectivity information provided by PhysicalProp messages. The relation $\text{linking}_P$ is implicitly provided at each PhysicalProp message through the list connections. Each connection in the list connections has a unique name and the list of IDs of locations that share the respective connection. The Bigraph Driver creates a hyperlink in the bigraph for each connection in connections.

Figure 6.12 depicts the generation of a bigraph from physical properties produced by a network of vehicles and sensors. Note that the Bigraph Driver not only generated the parenting relation, e.g., the parent of uav0 is oilSpill, but also the linking graph, e.g., ais is defined as a link shared between uav0 and drifter0...N.
CHAPTER 6. CASE STUDY: OIL-SPILL MONITORING MISSION

Figure 6.12: Bigraph Driver example.

Bigraphs distribution

The Bigraph Driver implemented for the oil-spill field test uses the Communication Service to exchange observed physical properties between ros_vehicles. The Bigraph Driver hosted at each ros_vehicle generates a bigraphical estimate of the global bigraphical abstraction. The bigraphical abstractions “flood” over the network of ros_vehicles eventually converging to a distributed bigraph estimate which is also globally consistent. In order to solve ambiguities between observations of the same physical space by different robots we augment each physical property with a global time-stamp. We use GPS time-stamps. If the Bigraph Driver receives a physical property that was already observed, it disregards the old one and re-generates the bigraphical abstraction.

During our field test, the UAV operators at each ground station and the military personnel onboard of the Navy vessel had access to distributed bigraph estimates. This addresses Requirement 6.3. Since the number of machines involved in our field test was small it was easy to enforce global consistency. In order for the system to scale, one should rely on local consistency since it requires less communication. This remains as future work.

Example 6.1 shows the use of the Communication Service to extend the situation awareness of UAV operators during a manoeuvre for handing-over control authority between operators.

Example 6.1. Consider an UAV uav0 that starts a mission under the control of gcs0 and, at a given point, hands control over to gcs1 on-board of a navy vessel. Figure 6.13 depicts this situation. Each operator has a local and limited observation of the world. The operator on the vessel does not know where the UAV is located until the handover has been successfully completed. The use of the Communication Service allows the operators to have access to an
extended bigraphical abstraction. With this information, both operators have access to the location of the UAV before and after hand-over.

![Figure 6.13: Distributed bigraph example.](image)

**Generation of control commands**

The BRR Driver translates BRRs to control commands that can be executed by a given vehicle. This addresses Requirement 6.4. In order to synthesize the physical control commands, the BRR Driver must also have access to the physical interpretations of the nodes in the BRR. To derive the physical interpretation, the BRR Driver subscribes physical properties from `ros_vehicle`. For example, the generation of a waypoint command to move an UAV to a given destination needs the GPS location of the destination. Figure 6.14 depicts the execution of a BRR for moving an UAV to the oil-spill location. The BRR Driver generates a GPS waypoint command to the centroid of the polygon that defines the oil-spill. This command is subscribed by the Autopilot Plugin from the `ros_vehicle` instance that is responsible for managing the execution of the waypoint.

**Logical-space programs**

Next we present some bigActors used in our field test. The bigActor defined in Figure 6.15 specifies the behavior of the UAV. The bigActor requests periodically to move the UAV to logical locations, e.g., move `uav0` to `oilSpill10` location. A period of one minute is enforced
CHAPTER 6. CASE STUDY: OIL-SPILL MONITORING MISSION

Figure 6.14: BRR Driver example.

BigActor hosted_at uav0 with_behavior{
    control(MOVE_HOST_TO(searchArea0))
    observe(CHILDREN(PARENT(HOST)))
    loop{
        react{
            case obs.contains(oilSpill10) => control(MOVE_HOST_TO(oilSpill10))
            case _ => observe(CHILDREN(PARENT(HOST)))
        }
        Thread.sleep(60000)
    }
}

Figure 6.15: Logical-space program for observing and tracking the oil-spill.

by the command `Thread.sleep(60000)`\(^4\). Since the oil-spill moves over time, each execution of the same instruction at the logical level maps to a different instruction at the physical level. In other words, a new waypoint command must be generated each time the logical location moves its physical location. Without logical-space programming, the operator would have to specify the new waypoints in physical-space which is inconvenient and could lead to mistakes. With logical-space programming, the command is the same, e.g., `MOVE_HOST_TO(oilSpill10)`. The LSEE is in charge of ensuring that the waypoint has the correct physical coordinates.

The bigActor in Figure 6.16 observes the bigraph with a query `LINKED_TO(HOST)` and

\(^4\)The use of `Thread.sleep` is not recommended since it blocks the actor’s thread. One can create a non-blocking actor implementation for a periodic loop. Nonetheless, for the sake of keeping the example simple, we use `Thread.sleep`. 
displays the result. The bigActor also matches messages handover and emergency. The first results in a BRR handing over the control authority for uav0 to ground station gcs0 and the second to gcs1. Due to the absence of interfaces in the autopilot Software Development Kit (SDK), the control action HANDOVER was performed manually by the operator using the command and control interface provided by the UAV autopilot vendor. We implemented a simple Graphical User Interface that sends messages handover and emergency under the request of the UAV operator.

\[
\text{BigActor hosted_at gcs2 with_behavior} \{
\text{observe(LINKED_TO(HOST))}
\text{loop} \{
\text{react} \{
\text{case obs: Observation =>}
\text{display(obs)}
\text{observe(LINKED_TO(HOST))}
\text{case "handover" => control(HANDOVER(uav0,gcs0))}
\text{case "emergency" => control(HANDOVER(uav0,gcs1))}
\}\}
\}
\]

Figure 6.16: Code for bigActor handover.

Our operators were able to watch bigraphs evolve as the field test progressed. The logical abstraction proved particularly useful for UAV handovers, since it provided the operators with information to be aware of the UAV location and connectivity regardless of which ground station held the control authority. The Bigraph Driver provided local bigraph estimates that where globally synchronized between ground stations over the internet using the communication service. The prior practice was to watch the ground station screen provided by the autopilot vendor and only see the UAV when under the control of the ground station. Correct termination used to be ensured by radio communication between operators. This communication was discontinued as the military operators came to understand and trust the logical bigraph abstractions being computed.

6.5 Final remarks

This chapter discusses the software implementation for a oil-spill monitoring mission using the logical-space programming paradigm. The field test addresses specifically the use of vehicles and sensors for complementing satellite imagery in the quest to monitor illegal bilge dumping activities. The field test uses an UAV with a camera and drifters with AIS modems and GPS to monitor an oil-spill. The oil-spill is emulated by a Navy vessel dropping 100 kg of pop-corn 6 km south of the shore of Portimão, Portugal. This is a
small oil-spill of the kind that might be created by a large ship flushing its oil tanks. We demonstrate the use of the logical-space programming paradigm for specifying the spatial behavior of vehicles and sensors used in the field test. The software implementation is named the Logical-Space Runtime System and follows the logical-space computing semantics presented in Chapter 5. The software infrastructure was successfully tested using real vehicles and sensors. The implementation was particularly valued for controlling UAVs logically and leaving the generation of GPS waypoints to the runtime system. It also made a large operational impact on the handover of UAV control authority from one ground control station to another.

LSRS is a first attempt towards a more general runtime system addressing a larger number of different computing machines. Next we present some lessons learned from this experience. Bigraphs are a suitable model for modelling location and connectivity of vehicles and sensors. Nonetheless, shared locations are not a first-class citizen in pure bigraphs. We believe that, in situations where vehicles operate in shared environments, one might need to use a location model allowing shared locations, e.g., Sevegnani’s bigraphs with sharing [22]. The implemented BRR Driver was limited to the BRRs used in the field test. One must study more general approaches that work for a wide variety of control commands performed by different machines such as indoor robots and smartphones. We believe this requires the definition of a set of canonical mobility commands that work on many physical machines. The naming service and the communication service were not implemented with scalability as a requirement. For a scalable implementation one must study different protocols for providing logical names to logical locations. The communication service currently requires vehicles to be assigned with a unique IP addresses. This practice is restrictive since most internet providers hide mobile devices under NAT addresses. One must find more scalable addressing schemes to extend the use of LSRS to mobile devices such as smartphones.
Chapter 7

Conclusions

In this dissertation we investigate the how to bridge programs with mobile computing machinery. We propose the BigActor Model as a bridging model between programs and logical-space models. The BigActor model [1] is described in Chapter 3 and combines Hewitt and Agha’s Actor model [2] for specifying concurrent reactive programs with Robin Milner’s Bigraphical Model [3] for specifying the location and connectivity of the computing machines. The BigActor Model makes location and connectivity first-class citizens in distributed machines. This is analogous to another bridging model, the von Neumann machine, which makes first-class citizens of memory, instructions, and their sequentiality.

The BigActor Programming Language (BAL) is an implementation of the BigActor Model for logical space programming. It has a runtime system named the BigActor Runtime System (BARS). The BARS targets an abstract machine (bigraphs). The abstract machine has to be realized on a physical space of mobile and distributed computing machines. The realization is produced by the Logical-Space Execution Engine (LSEE), which bridges bigraphs with the physical space. The Logical-Space Runtime System (LSRS) extends BARS with LSEE so that programs written in BAL can seamlessly execute over physical spaces. The second part of this dissertation is concerned with the formalization and implementation of the interactions between logical spaces and physical spaces.

First, we approach this problem formally, by introducing the logical-space computing semantics. This is presented in Chapter 5. In logical-space computing, spatial agents operate over logical-space models while the runtime system is in charge of interacting with the physical space. In Chapter 6 we presented an implementation that follows the logical-space computing semantics of Chapter 5. The LSRS uses the LSEE to generate logical-space models using bigraphs. The physical space is modelled using polygons defined using GPS coordinates. The spatial agents are bigActors. Our implementation programs robots and sensors in logical-space to execute an oil-spill monitoring exercise in the Atlantic. BigActor programs execute over BARS, which interacts with physical spaces through the LSEE. LSEE executes over the Robot Operating System (ROS) - an open-source middleware for robotics. The physical machinery used in the demonstration consisted of one Air Force UAV, three ground control stations, four drifters that broadcast their position using AIS, and one Navy
vessel equipped with a small speedboat. The Portuguese Navy emulated the oil-spill by releasing 100kg of popcorn in the ocean.

As future work we want to investigate the use of specific formalisms for modelling and controlling physical spaces. The logical-space computing semantics defines physical spaces abstractly and our implementation uses polygons as physical space models. One would benefit from the definition of a concrete physical-space model that can still be useful for a large spectrum of mobile computing applications. This is particularly interesting for the definition of mobility commands that can be transparently used for controlling heterogeneous mobile computing machines, e.g., an autonomous vehicle or a person with a smartphone.

Bigraphs are a suitable model for the location and connectivity of vehicles and sensors. Nonetheless, shared locations is not a first-class citizen in pure bigraphs. We believe that, in situations where vehicles operate in shared environments, one might need to use a location model that allow the model of shared locations. This problem can be addressed by extending the formalism to use Sevegnani’s bigraphs with sharing \cite{22}.

Our implementation of the Logical-Space Runtime system addresses our case study. We want to build the implementation to program a large spectrum of mobile computing applications. We envision the need for a scalable naming service. A possible approach is to adopt a standard naming service such as Common Object Service (COS) Naming from CORBA, Domain Naming System (DNS) from the internet, and Java Naming and Directory Interface (JNDI) from Java.

Our implementation requires computing machinery to have an unique IP addresses. This practice is restrictive since most internet providers hide mobile devices under NAT addresses. One must find more scalable addressing schemes to extend the use of LSRS to mobile devices such as smartphones.


Bibliography


[90] Marin Chintoan-Uta and Marin Chintoan. Integrated maritime safety & surveillance systems integrated maritime data environment (imdate) platform= emsa capability to provide real time integrated maritime information services.


