Bigraph Metaprogramming for Distributed Computation

A thesis

submitted in partial fulfilment

of the requirements for the Degree

of

Master of Science

at the

University of Waikato

by

Gian David Perrone

THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

University of Waikato

2010

Abstract

Ubiquitous computing is a paradigm that emphasises integration of computing activities into the fabric of everyday life. With the increasing availability of small, cheap computing devices, the ubiquitous computing model seems more and more likely to supplant desktop computing as the dominant paradigm. Similarly, the presence of high-speed network connectivity between vast numbers of computers has already made distributed computing the preferred paradigm for many application domains. Unfortunately, traditional approaches to software development are not necessarily well-suited to developing software in a post-desktop world. We present an extension to the bigraphical reactive systems formalism that enables us to construct a programming language based upon it. We believe that this programming language provides programmers with an environment better suited to the challenges that arise when creating software within a distributed or ubiquitous computing paradigm. We detail our modification to the theory of bigraphical reactive systems that enables metaprogramming. Finally, we provide a description of our prototype implementation of a programming language that enables metaprogramming of bigraphical reactive systems.

Acknowledgements

I would like to gratefully acknowledge the support and advice offered by my supervisor, Dr. David Streader during this project, and throughout my university career.

To my partner Rocky Maeva, thank you for all of the love, support and encouragement that has made this possible.

Finally, I would like to thank Anthony Blake, Alyona Medelyan, Perry Lorrier, my parents John and Vye Perrone, Professor Steve Reeves, and Associate Professor Thomas Hildebrandt, all of whom have been willing to engage in stimulating conversations and read drafts throughout this project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Ubiquitous (or *pervasive*) computing, in which many small, connected devices "disappear" into the fabric of everyday life, has been a dream of those working within computing disciplines since the late 1980s [40]. By combining many such devices, a user need not be aware that he or she is interacting with a particular computer. Instead, the sum of the behaviours of many connected elements provides the desired behaviour based upon the location of the user, or some other measured quantity from the physical environment. Such ubiquitous computing can mediate social interaction, provide context-aware services, and dynamically "coalesce" appropriate computing resources required to perform some task in response to a user action.

The increasing availability of commodity low-power computing devices and high-speed network connectivity has given rise to a situation in which the departure from the traditional "one user—one desktop" model appears viable. Since the invention of mobile devices such as mobile phones and PDAs with previously unimagined computing power it is now common for a single application to be running tasks simultaneously in many locations—e.g. on another machine connected via the internet, on a mobile device, or within a "cloud" of computing resources—while still providing the appearance (to the user) of a single, cohesive application. However, while the departure from the desktop model of computing is feasible from a technical perspective, the software de-

velopment tools and techniques needed to manage this increased complexity have not developed at the same rate as the hardware infrastructure that makes it possible.

In attempting to provide a unifying approach to the various means of creating distributed and ubiquitous software systems, we present a programming language based upon *bigraphical reactive systems* [26, 29]. The language we present is one in which *processes* (or *agents*) that express some computation may be exchanged between hosts and executed in a secure manner to achieve computation that scales to exploit available distributed computing resources, while reducing the quantity of *dedicated* computing resources that are required to operate a service with rapidly changing usage.

We propose (as an extension to previous work on *bigraphical programming languages* [7]) a single mechanism that allows the user to express both computation and high-level system and agent behaviour by exposing a specialised form of bigraphical reactive systems within a programming language. We believe that such a programming model is well-suited to both current challenges within *mobile* and *distributed computing*, as well as being an appropriate foundation for future development of ubiquitous computing solutions.

Bigraphs are a recent modeling formalism introduced by Milner in [28]. Bigraphical reactive systems include *reaction rules* [28, 27] that imbue them with dynamic behaviour, permitting description of runtime changes in the locations of processes and enabling processes in different locations to communicate with each other, making bigraphical reactive systems an ideal formalism upon which to build the kinds of mobile, distributed computation that we wish to achieve.

1.1 Distributed Computation

Distributed computation is a model in which multiple independent computing resources communicate in order to achieve some computational goal. In real terms, this usually means multiple interconnected computers running programs

that are either too large to run on a single computer, or those which achieve performance increases when running some parts of the process in parallel. Communication generally takes place as some kind of message passing, and memory is not shared between nodes (such a system is therefore often called a *distributed memory* system, in contrast to a *shared memory* system). This kind of distributed computing is already a popular computing model that appears to share many of the qualities of the kinds of *ubiquitous* systems that will be enabled by the ever-increasing quantities of small, network-attached devices.

1.2 Context-Aware Computing

Context-aware computing may be viewed as one of the most immediately usable applications of the ubiquitous computing model. Context-aware computing is concerned with the creation and analysis of software that uses information from a user's surroundings to perform some context-sensitive behaviour [30]. This "user context" may be a simple abstraction of some physical location, or it may be a sophisticated representation based upon real-time sensor information enabled by the presence of many small, network-attached devices. Traditional programming methodologies provide a poor model for this kind of interaction [43]. A program needs to have both the ability to express statements about the user's context (e.g. location, time of day) as well as the "IT context" in which it exists (e.g. network connectivity, presence of other hosts, services available) [13]. While previous attempts at directly modeling both contexts as bigraphical reactive systems have been dismissed as "awkward" [7], we show that with a modification to the encoding of such systems presented in Section 3.5.2, this awkwardness can be alleviated and systems can be modeled in a way that more closely matches the programmer's intuitions.

1.3 Intelligent Agents

While the traditional view of distributed computing is that of its role in high-performance or scientific computing, the rise of ubiquitous mobile devices and persistent high-speed internet connections has led to a new range of applications that are a natural fit for both the existing distributed computing idiom and the emerging ubiquitous computing idiom. With the ability to exchange large volumes of data at high speed, it is no longer important that the user and his or her computation be geographically (or topologically, in network terms) close. Processes can be started wherever the data or resources required to run a program are available. With flexible provisioning services (such as *cloud computing*), it may make sense to reconfigure the way in which a computation is distributed in order to minimise the cost, or the make use of additional resources that become available.

This connected, dynamic environment leads to a class of systems known as *intelligent agents*. An agent is a network-aware process that may move between systems in order to act on behalf of a user [37]. A software agent may have partial or total knowledge of the network environment within which it exists. While there exist many definitions in the literature (e.g.[37, 14, 36]), there appears to be little consensus on exactly which qualities define an agent. Consequently, we suggest four properties that define an agent:

- Some computation to be performed in the presence of suitable input.
- A current state, corresponding to the computation that has been performed previously by the agent (in AI terms, this may be known as the "mental state" of the agent [37]).
- Some model of the network environment (partial or complete), including the current location of the agent within the network topology.
- An ability to communicate with other agents or services in the same location, possibly including an ability to request that the current location

relocate the agent to another location.

We show in Section 3.8 that there exists a natural encoding of all of these properties as *bigraphical reactive systems*.

1.4 Services

Included in our definition of an agent is some notion of a *service*. Whereas an agent is network-aware and inherently *mobile*, a service is location-specific and does not need to be network-aware. It needs only perform some function when requested to do so by an agent. For example, one might construct an agent designed to collect files of a certain type stored on some network. The agent might arrive at a location, request from the *File* service a list of all files at that location, and then request all of the files matching its internal criteria. The agent would then request from the *Network* service that it be moved to some other location with a different *File* service. We provide a more formal treatment of services in Section 4.3.

1.5 Parasitic Computation

Parasitic computation is a technique first described in [4] that uses the legitimate function of communicating hosts to perform some other (user-defined) computation. In [4], the behaviour of the TCP/IP (Transport Control Protocol/Internet Protocol) sub-system of a remote computer is exploited in order to perform computations (in this case the factoring of keys used for encryption). When a packet of information is sent across the internet, the host receiving that packet performs a *TCP checksum* on the data to ensure that it has arrived intact. If the packet checksum fails, then a message is sent back to the host to indicate that the packet needs to be re-transmitted. By crafting packets in a particular way, it is possible to make the success or failure of the checksum equivalent to the result of a particular computation. By encoding a computa-

tion into many packets and sending these to any available hosts, it is possible to distribute a computation in a way that makes it very difficult for a user to even detect that their computer is being used to run someone else's computation. While there are ethical and legal implications to this, assuming that the user does consent, the ability to distribute very small units of work to many different computers without needing to authenticate or be concerned with the topology of the network is very appealing in some distributed applications.

It is this ability (of parasitic computing) to enable small computations to be performed in untrusted distributed environments without manual intervention that we aim to replicate with our system. However, we provide a more expressive normative means for describing any computation, rather than crafting a parasitic solution to a particular computational problem.

1.5.1 Parasitic Javascript

Parasitic *Javascript*, introduced in [18] follows the model provided by the work in [4], however instead of embedding computation inside TCP/IP packets, it embeds it into web pages.

Many modern websites employ the Javascript language in order to provide some dynamic functionality within a page. Javascript is supported by almost all modern web browsers, and is executed in a "sandbox" environment on the *client-side*, i.e. the code is downloaded from the remote server and then executed locally. One feature of Javascript is its ability to construct "AJAX" requests that allow data to be passed back and forth between the server from which the code was downloaded and the client computer on which it is being executed. By embedding a piece of Javascript code inside a web page that communicates the results of that computation back to the server it was downloaded from, one can perform computation that is distributed across all of the computers being used to view that web page.

While Javascript provides a general-purpose scripting language in which to perform computations, the browser environment is not necessarily ideal

for distributed computation. The browser security model for Javascript is designed to be particularly restrictive, and Javascript programs that are using a large number of resources may be terminated by the browser to prevent them from adversely affect the operation of the rest of the browser (as most browsers are single-threaded, and therefore the scripts they run are not subject to the normal operating system-level scheduling). Similarly, Javascript is not "network-aware" by default, and so any parasitic Javascript program must be manually constructed by the programmer in a way that is tailored to the parasitic execution model.

We suggest a means (in Section 6.2) of integrating our proposed computational model with existing approaches to parasitic Javascript.

1.6 Related Work

1.6.1 Bigraphical Programming Languages

The BPL (Bigraphical Programming Language) project at the IT University of Copenhagen has focused on the use of bigraphical reactive systems for modeling and construction of so-called *context-aware systems*, in which information about the environment (from sensors) is available to guide the execution of *location-aware* software [7]. In exploring approaches to modeling context-aware systems as bigraphical reactive systems, the authors found that the naive approach to modeling was "somewhat awkward" [7], and instead propose an alternative modeling technique that they call *Plato-graphical models*.

Plato-graphical models provide separate encodings of the actual context in which an agent exists, and the agent's representation of that context. Similarly, within this technique reaction rules are no longer used to directly describe computation or mutation of state in the system, but rather are used to encode operational semantics of programming languages, which one can then use to construct programs that operate over representations of the context in which the programs operate, and to encode transitions occurring in the real world

[7].

The authors cite three reasons for the awkwardness of the direct-modeling "naive" approach, which leads them to reject it as a means of encoding context-aware systems as bigraphical reactive systems:

1. Bigraphical reaction rules do not support recursion directly, requiring implementation (in bigraphs) of some sort of runtime call stack using controls on nodes.

2. Bigraphical reaction rules that model *queries* over contexts (a fundamental operation in context-aware computing [40, 37]) will apply over *any* context, due to the interpretation of reaction rules within the theory of bigraphical reactive systems. This can only be avoided through the secondary implementation of some kind of *program counter*.

3. Reaction rules cannot easily encode the case in which a rule should only apply in the *absence* of something.

Birkedal et al. assert (in [7]) that the naive approach to modeling context-aware systems within bigraphical reactive systems is unsuitable without modification. We believe that the direct modeling approach remains valuable for many kinds of modeling tasks, and consequently we provide a modification to the structure of bigraphical reactive systems in Section 3.5.2 that we believe addresses the problems discovered within this approach by the BPL project.

1.6.2 Mobile Processes

While modeling formalisms such as CSP [16], CCS [22] and Petri Nets [32] have been employed in academia and industry extensively, a relatively recent extension has been the introduction of *process mobility*, most notably within the π-calculus [23]. The π-calculus enriches normal process calculus-style constructs with the ability for processes to change location and replicate. Communications channels are first-class values, and so may themselves be commu-

nicated (via channels) between processes. This increased expressivity is ideal for modeling mobile processes.

1.6.3 Pict/Executable π-calculus

The development of the π-calculus has led to the development of languages such as *Pict*, that is essentially an executable form of the π-calculus. Pict has been designed as a concurrent programming language for expressing the behaviour of groups of processes that communicate and interact to perform some computation [33]. It has been suggested that it might be an ideal language to embed within bigraphical reactive systems in order to provide a language with which to express low-level computation [7].

1.6.4 Evaluation by Graph Reduction

Several functional programming languages have used graph rewriting to provide evaluation semantics for programs, including SASL [39], Clean [8], Lazy ML [2] and Haskell [17].

We draw some inspiration from these works, as well as the early work by Barandregt [5] on term graph rewriting in designing our computational sublanguage for expressing computations within bigraphs.

Chapter 2

Bigraphical Reactive Systems

Bigraphical reactive systems are a class of formalisms in which bigraphs are enriched with dynamic behaviour through the introduction of *reaction rules* [28].

2.1 Pure Bigraphs

At the core of any bigraphical formalism is the class of bigraphs known as *pure bigraphs* [29]. These pure bigraphs contain no notion of name-binding or of *reactions*, i.e., they exhibit no dynamic behaviour. A pure bigraph may be characterised formally by the definition (from [29]):

$$G = (V, E, ctrl, G^P, G^L)$$

$$G^P = (V, ctrl, prnt)$$

$$G^L = (V, E, ctrl, link)$$

Where V is the set of nodes and E is the set of edges. *cntrl* is the *control map*, $V \to K$, that associates a *control* (from the set of controls K) with every node in the bigraph. G^P is the *place graph*, while G^L is the *link graph*.

A control is a kind of "signature" for a node, defining its arity (i.e. the number of connection points, or *open links* it has to connect it to other nodes)

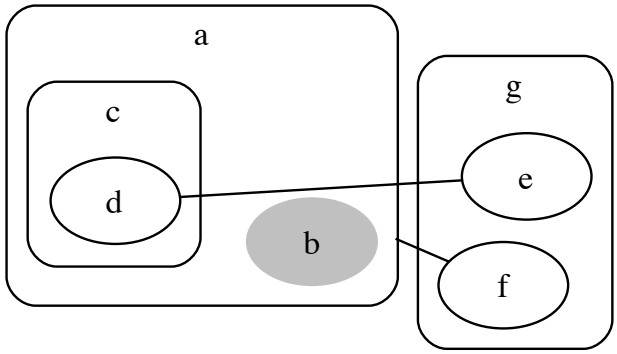


Figure 2.1: A pure bigraph

and within pure bigraphs, defining a node to be *active* or *passive*. As we extend our definition of bigraphs, more information will be added to the controls that we associate with nodes, such as type information.

A place graph (G^P) is a set of possibly-nested nodes. The *parent map* *prnt* captures this notion of nesting, by mapping each node to a parent (every node is contained within exactly zero or one parent node). The place graph is designed to capture notions of *location* within a system. A node might correspond to some built location (e.g. a building or a room), or it might represent some abstract notion of location (e.g. a host or a set of nested parentheses in an expression).

A link graph (G^L) is an undirected hypergraph that shares a set of nodes V with the place graph. The link graph ignores the nesting structure of the place graph and may connect any nodes together (as defined by the *link* function). The link graph is designed to capture notions of communication and connectivity - e.g. a network link, a physical wire, a shared name, or having some other ability to communicate or be considered as a group.

We have omitted the notion of *inner* and *outer faces* usually given in a treatment of bigraphs, as they are not used within our eventual application of bigraphs. Instead we employ *named ports* that are less expressive than the definition of interfaces given in the literature. Indeed, our named ports (discussed in Section 3.2) could be implemented within the usual bigraph interfaces presented in [29].

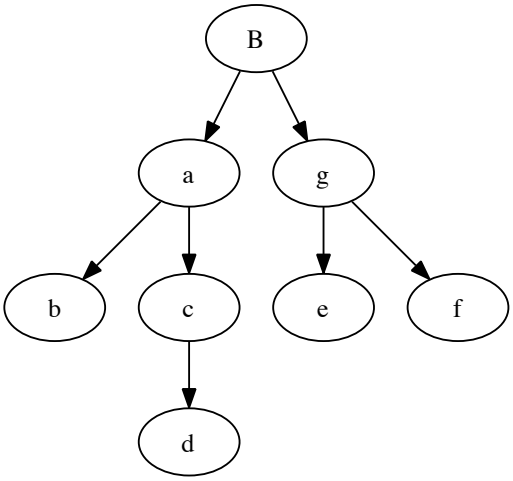


Figure 2.2: The place graph for the bigraph in Fig. 2.1 with implied *frame B*

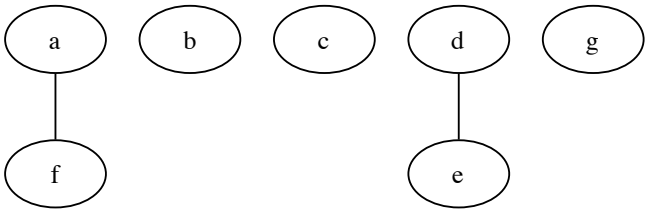


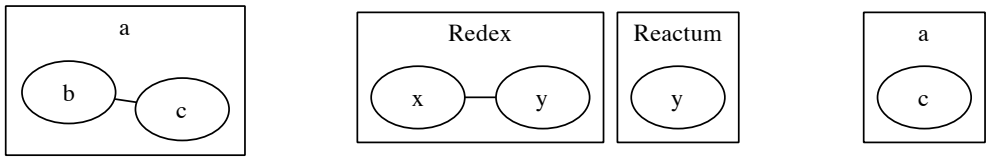
Figure 2.3: The link graph for the bigraph in Fig. 2.1

Fig. 2.1 demonstrates a pure bigraph, while Fig. 2.2 and Fig. 2.3 show its decomposition into separate representations of the place and link graphs. Fig. 2.1 also includes a *site* (also known as a *hole* in some presentations of bigraphs) that is represented with grey shading. This is a hole into which another bigraph may be substituted, or to express the notion of matching *any* node as the formalism is extended to include notions of pattern-matching and rewriting.

2.2 Bigraphical Reactive Systems

While a pure bigraph captures useful notions of static models, to imbue our models with dynamic behaviour we use an extension of pure bigraphs called *bigraphical reactive systems* (BRS). These BRSes extend bigraphs with a notion of *reaction*, which is essentially *graph rewriting* on bigraphs.

Reactions are expressed in the form $r \rightarrow r'$, where r is known as the *redex*



(a) A pure bigraph G (b) A reaction rule $r \rightarrow r'$ (c) G'

Figure 2.4: An example bigraphical reactive system

and r' the *reactum*. The redex defines a pattern to match against some bigraph, and (in the presence of such a match), the reactum defines the rewriting to perform upon the matching sub-graph [28].

Fig. 2.4 demonstrates a pure bigraph (Fig. 2.4a) being enriched with a reaction rule (Fig. 2.4b), and finally the modified graph after the reaction rule has been applied to G (Fig. 2.4c). It is worth noting from the figure that reaction rules are themselves bigraphs, which gives rise to a natural *homoiconicity* property, in which bigraphs may be used to manipulate bigraphs (including themselves). We exploit this property in order to permit recursion (Section 3.7.2) and to develop an exceptions mechanism for our Lope bigraphical agent programming language (Section 3.7.1).

2.3 Process Calculi Embeddings

The bigraphs formalism was developed to be a unifying formalism for the myriad of process calculi that are used currently in modeling tasks, as well as to provide a more powerful mechanism for specifying the types of complex ubiquitous computing systems that are increasingly common [28].

Encodings have been demonstrated for Petri Nets [24], π-calculus [9], λ-calculus [25] and others. The range and diversity of the formalisms that have been shown to be representable within bigraphs provide some evidence for the expressive power of bigraphs for modeling systems.

2.4 Modelling with Bigraphs

While the traditional approach to bigraphs has been to use them as a meta-formalism for reasoning about process calculi and other modeling formalisms, there seems a natural mechanism for modeling directly within bigraphs. By providing appropriate syntax and a default set of reaction rules within a programming language, it is possible to expose to a programmer a language that has many of the desirable properties of a "modelling" formalism, while still being able to directly express efficient computation in the manner of a *programming* language. We describe our approach to the design and implementation of such a language in Chapter 3.

We also demonstrate a means of encoding reaction rules within bigraphs themselves in order to reduce the awkwardness associated with modelling directly within bigraphs, as observed by Birkedal et al. [7] and described in Section 1.6.1. We show (in Section 3.5.2) that a simple modification to the encoding of reaction rules within bigraphical reactive systems enables queries over limited contexts and a form of metaprogramming (discussed in Section 3.7) over bigraphs that permits general recursion.

2.5 Embedding Computation

As a very general formalism, bigraphs provide no fixed mechanism by which actual computation should be encoded. Consequently, there are a number of strategies that may be employed to permit the description (and execution) of computations within bigraphs. We therefore present in the remainder of this section a brief survey of several approaches to embedding computation within bigraphs that would permit convenient direct encodings of low-level computation.

2.5.1 Mini-ML

Mini-ML [11] was embedded within bigraphs in order to facilitate context-aware modelling using the Plato-graphical models described in [7]. This Mini-ML is essentially a typed λ-calculus with references and side-effects. The natural numbers were defined using an encoding similar to Peano arithmetic (i.e. a *zero* symbol and a successor function).

The ability to encode a general-purpose programming language such as ML [21] into bigraphs demonstrates that there should be no significant barrier to encoding many other kinds of programming languages within the bigraphs formalism.

2.5.2 Pict

With pre-existing encodings of the π-calculus within bigraphs [9], it seems likely that Pict [33] (which is based upon the π-calculus) could be encoded within bigraphs either by translation to bigraphical π-calculus primitives, or through implementation of the *Pict Abstract Machine* [41] within bigraphs (either directly or implemented within some other encoding of computation).

2.5.3 Clean

Clean [8] is a general-purpose functional programming language that is executed by graph rewriting (i.e. functions are rewrites on some graph). It is conceivable that these graph rewriting rules could be implemented using the same mechanism used to perform graph rewriting within bigraphical reactive systems. Similarly, an implementation of the *ABC* machine [34] (an abstract machine designed to execute Clean programs) within bigraphs could provide another means of directly executing Clean within bigraphs.

2.5.4 Java

From a purely pragmatic view, it may be desirable to allow execution of arbitrary programs written in other languages without reference to the bigraphical idiom. It might therefore be possible to generate adapter code that would allow Java code to be loaded as a first-class value inside a bigraph node, and then in the presence of suitable input to that Java program, a step of "reduction" takes place that rewrites the code and input to the output of that program.

This approach might allow the kinds of context-awareness and process mobility that the bigraphical programming languages model enables to be extended to existing programs, running without modification inside a mobile, context-aware system.

Chapter 3

A Metaprogrammable Bigraphical Programming Language

In this chapter we present a design for a bigraphical programming language for ubiquitous and mobile computing, called *Lope*, based upon bigraphical reactive systems, as described in Section 2.2. However, whereas bigraphs are primarily a modeling formalism (most often applied to the embedding of other process calculi) the language we present is distinctly a *programming* language. Consequently, where theoretical considerations from the theory of bigraphs and practical considerations come into conflict, the pragmatic route has been taken. Similarly, we provide features for both programming-in-the-large (i.e. for describing the structure of the system and organising program elements) as well as a computational sub-language for actually expressing computation in a way that would be familiar to users of many modern functional programming languages (such as Haskell or ML). Existing approaches to bigraphical programming languages have traditionally been more concerned with system-level modeling and encoding programming language semantics, rather than with describing computation and modeling directly within bigraphs [6, 7].

3.1 Program Structure

A Lope program consists of the following elements:

- A *place* graph, consisting of one or more (possibly-nested) nodes
- A *link* hypergraph, consisting of zero or more links (each of which may link two or more nodes, shared with the place graph)
- A set of *reaction rules* that describe valid bigraph rewritings

This definition is consistent with the definition of bigraphical reactive systems given in [25].

3.1.1 Syntax and Graphical Representation

We define a syntax for describing Lope-style bigraphs, however our approach follows that taken in [24], in which it is stipulated that the graphical representation of bigraphs should be considered primary. The Lope syntax simply provides a convenient means of describing Lope-style bigraphs, however we will continue to use graphical representations of systems and reaction rules in addition to the programming language syntax. We hope that some future extension of this work would permit the direct construction of Lope programs within some visual environment.

3.2 Named Ports

The link graph within our Lope-style bigraphs is meant to encode information about connectivity or ability to communicate, however these links are not used for actual communication (i.e. they are not necessarily channels in the traditional distributed systems sense). While reaction rules could be designed that only move child nodes between parents connected by a certain type of link in order to represent the transfer of data over network links (and indeed, in a distributing computing situation this would be highly desirable), this is

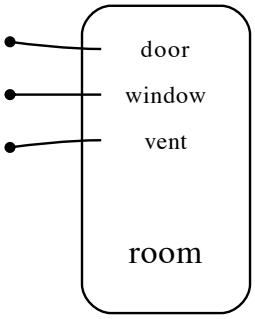


Figure 3.1: A node with several named ports

not enforced by the language or the environment, but rather is a convention that a user could define for a given program.

In contrast to the usual notion of *interfaces* in bigraphs [24, 25, 26], we opt for a much simpler mechanism that is more appropriate to the programming language motivation behind our language design. A *port* is a named connection point for links. Only one link may connect to each port, though these links are hypergraph edges, and so they may link multiple nodes together. Self-loops that link two ports of the same node together are permitted (however the same restriction applies such that a link may not connect the same port to itself).

Fig. 3.1 provides an example of a node with several named ports. The names are not semantically significant and have no special meaning, however they can be matched against in reaction rules (e.g. match any node connected to any other with a port named "door"). Similarly, ports may have *types* (which must agree between the connecting ends so as to establish a single type for the link) that may also be used to distinguish nodes and links while matching (e.g. match any two nodes connected by a link of type "network").

3.3 Syntax

The design of the Lope syntax follows the principle that a Lope program is simply a description of a bigraph. Only in the computational sub-language is there deviation that introduces syntax specifically used to describe groups of

nodes and links.

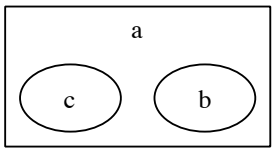
The creation of a node uses the syntax:

$$node ::= identifier[(sites)][< ports >][: type][:: Kind]\{[nodelist]\}$$

Square brackets indicate that the inclusion of a piece of syntax is optional. The simplest node we could therefore construct would be:

```
a {}
```

This constructs an empty node called a. We can construct nested nodes in the same way, by including additional node definitions inside the body of the parent.



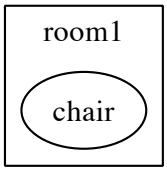

```
a {
    b {}
    c {}
}
```

Figure 3.2: A bigraph with one-level nesting

Fig. 3.2 demonstrates a node a with two empty children, b and c. These nodes have no sites or ports, and there are no reaction rules defined within the scope of our system. To create a more complex model, we can start to introduce these elements:

```
room1($x) {
    $x
}
```

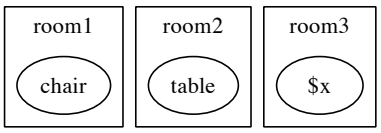
This example defines some node $room1$ that has a single site $\$x$. However, because $room1$ has a site, we need to introduce a notion of *substitution* and *assignment*:




```
val c = chair {}

room1($x) { $x }

room1(c)
```

The *val* syntax introduces a node, but does not add it as a child of the current parent node (in this case, the implied *world* node). Instead, it introduces a name within the current scope that can be used to refer to that node. Consequently, the example above first creates (within the current scope) a *room* node with a site called $\$x$, and then inserts into it a *chair* node. There is no *chair* node created within the parent scope, and actual instantiation of the *chair* node only occurs at the application $room(c)$. This corresponds to a kind of *lazy* evaluation of the right hand side of the *val* binding, such that all references to c are replaced with the body of the val binding.

We wish to generalise this notion of parametrisation, and therefore introduce a *template* keyword:




```
template room($x) {
    $x
}

room1 = room(chair {})
room2 = room(table {})
room3 = room(_)
```

This example introduces three *room* nodes to the current scope named *room*1, *room*2 and *room*3. The *room*1 node contains a node *chair*, while *room*2 contains a *table* node. A new piece of syntax is introduced — _, the *wildcard* operator. This indicates that the site should not be substituted with anything, and should instead remain unbound. Consequently, *room*3 remains empty, however we could still place an element into it at some later time.

3.3.1 Type and Kind Annotations

Implicit in our example using the *template* keyword is the assumption that the three *room* nodes we instantiate have the same structure, and indeed the same *type*. This is indeed the case, as the *template* keyword silently introduces a new unique type that represents the type of all instantiations of that template.

We can make this explicit:

```
kind Furniture
kind Chair

type room
type chair :: Furniture, Chair
type table :: Furniture

template room($x :: Furniture) : room {
    $x
}

room1 = room(armchair : chair {})
room2 = room(table : table {})
```

The *kind* keyword introduces a new kind (type of type) named $Furniture$ to the current environment. Similarly, the subsequent *type* declarations introduce new types. The single colon (:) is used to denote that a node has a certain type, while the double colon operator (::) indicates that a type belongs to a kind. The result of these declarations is that the *room* template will now only accept substitution of nodes into the site $\$x$ that are of kind $Furniture$, and will return a node of type *room*. Because we have not defined a kind for the *room* type, it implicitly inhabits only one kind, the built-in $Node$ kind.

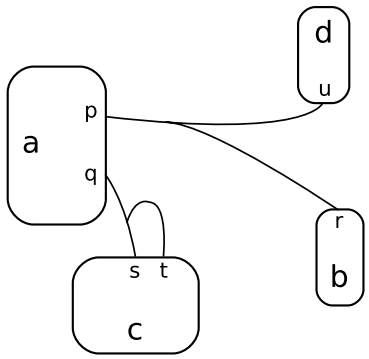
This kind of judgement starts to make strong statements about the structure of nodes, and we may begin (through type checking) to reject certain invalid programs. For example, we could statically detect the case in which one tried to substitute a node of type *room* into another of type *room*, because *room* is not of kind $Furniture$.

Nodes that have no type annotation will be assigned an automatically generated type that is unique to that node, and which is only of kind $Node$.

3.3.2 Link Syntax

The primary means of introducing links is through the *link* keyword and the linking operator $< - >$. The exact use of this is slightly complicated by the fact that the link graph within Lope (as within all bigraphs) is a *hypergraph*,

in which a single edge may connect two or more nodes:




```
a <p,q> { }

b <r> {}

c <s,t> {}

d <u> {}

link a.p <-> b.r <-> d.u

link c.s <-> a.q <-> c.t
```

There are two distinct link statements made in this example. The first *link* keyword introduces a hypergraph edge linking the ports $a.p$, $b.r$ and $d.u$. The second introduces a second edge linking $c.s$, $a.q$ and $c.t$. This self loop is permitted. Order is not important, as link graph edges are undirected. The effect of defining two links involving the same port is to merge those two edges into a single hypergraph edge.

In the absence of types, a unique kind is inferred for all of the ports connected to a given edge. For example, the first link made in the example above might result in $a.p$, $b.r$ and $d.u$ all being given the kind $Uniq001$ and the second set being given the Kind $Uniq002$. In the presence of explicit type information, checking will be performed to ensure that there is agreement between the types and kinds of the ports connected by a single link, and a type error will result if this agreement does not exist.

Where port names are the same, it is possible to use the *link* keyword without port names specified:

```
building <staffDoor,publicDoor> {}

fireEscape <publicDoor> {}

link building <-> fireEscape
```

In this case, the *link* keyword will have the effect of linking ports with the same name, resulting in the bigraph in Fig. 3.3. This is simply syntac-

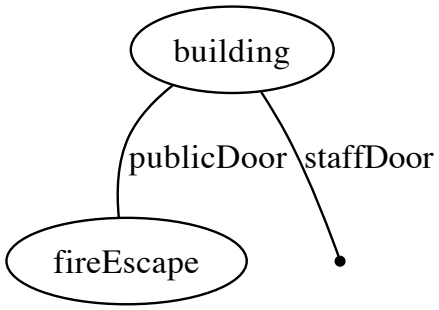


Figure 3.3: The bigraph resulting from use of the *link* keyword

tic sugar for `link building.publicDoor <-> fireEscape.publicDoor`, and the `staffDoor` port on the *building* node will remain unlinked.

3.4 Types and Kinds

All nodes within our Lope-style bigraphs have some *type* associated with them. This corresponds to the usual notion within programming languages of values having types. However, we also assign *kinds* to our types (i.e. types of types). We use lower-case names for types (e.g. *string*) and capitalised names for kinds (e.g. *Process*). For example, a node representing the integer value 32 would have the type *int*, and the kind $Value$, which distinguishes it from some computation that may eventually yield an *int* after some number of reaction rules have been applied. A single type may inhabit multiple kinds, so while a node has exactly one type, it may have multiple kinds.

The inclusion of kinds within Lope was necessary to control the order in which low-level computation is evaluated with respect to the rest of the system. Without being able to readily distinguish computation from reaction rules operating on other kinds of nodes it is difficult to implement a predictable evaluation order consistent with the normal interpretation of mathematical operators, because of the *fairness* property discussed in Section 3.7. Similarly, kinds provide a convenient means for users to construct reaction rules that operate in limited contexts without needing to duplicate rules for every type of node that is of interest within that context.

It is important to distinguish our Lope programming language notion of "kinds" from the use of the same term in the literature (e.g. [31]). We use the term "bigraphical kind" to refer to the definition of kinds within *kind bigraphs* [31], where kinds are used to restrict valid nestings of place graph nodes, and "Lope kind" to distinguish our use of types-of-types within our programming language. The latter use corresponds more closely to the definitions used in programming languages such as Haskell, in which their *type classes* mechanism is essentially a means of making kind-judgements. Where no ambiguity exists, we will always use "kind" alone to refer to the Lope- and Haskell-style *kinds*. Similarly, we will often refer to a node being of kind K. This is informal usage, meaning that the node is of a type that is of kind K.

Lope kinds may be parameterised by types, type variables or other kinds that indicate the presence of "sites" within nodes of that kind, and the types (or kinds) of the bigraphs that may be substituted into those sites. A kind with uninstantiated parameters is called a *kind signature*.

Similarly, types may be parameterised by other types or type variables, but not by kinds. This corresponds to a much more traditional notion of types in a strongly-typed programming language such as ML or Haskell.

3.4.1 Typing Rules

The type judgements used to establish whether a given term is well-typed are given below, starting with ground terms. We make reference to the type environment Γ, and the kind environment Γ_K.

3.4.1.1 Ground Terms

Some primitive objects within the language (such as literal values) may be assigned a type without the need for further inference. These correspond to the axioms in our type system:

$$\frac{t \in \mathbb{Z}}{\vdash t : int}$$

For strings, we use regular expression notation to indicate the matching of all strings enclosed by quotation marks:

$$\frac{t = \text{“}\Sigma * \text{”}}{\vdash t : string}$$

$$\frac{t \in \{true, false\}}{\vdash t : bool}$$

3.4.1.2 Un-typed Nodes

Because the Lope syntax provides the capability to construct nodes that have no explicit type information, there exists a judgement for un-typed nodes that assigns it a unique type:

$$\frac{t = n : \_, u :: Type, u \notin \Gamma}{\Gamma \vdash t : u}$$

3.4.1.3 Templates

Template declarations are instantiated with explicit or implicit type information, and are typed as functions over nodes:

$$\frac{s = template(x_0 : t_0, x_1 : t_1, ..., x_n : t_n) : r}{\Gamma \vdash s : t_0 \to t_1 \to ... \to t_n \to r}$$

Similarly, template instantiation enforces type equality (or kind-inhabitation) on parameters and arguments:

$$\frac{e = identifier(v_0 : t_0, v_1 : t_1, ..., v_n : t_n) : t'_0 \to t'_1 \to ... \to t'_n \to r}{\Gamma \vdash e : r, t_0 = t'_0, t_1 = t'_1, ..., t_n = t'_n}$$

3.4.2 Kinding Rules

The presence of *kind* declarations within a program assigns types to kinds. A type may be of more than one kind, and will be a member of at least one kind

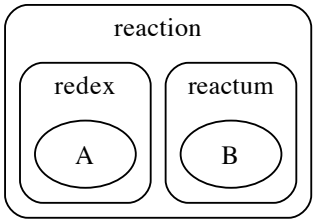


Figure 3.4: A reaction rule

(the base *Node* kind). We define the kind environment Γ_K, and a predicate $kind(type, kind)$ that provides a kind judgement that some type is of a kind.

Kind inclusion is therefore determined by instantiation of kinds within the kind environment through the use of the *kind* keyword, by inhabitation of kinds through the use of annotations on the *type* keyword, and by implied inhabitation of the *Node* kind by all types:

$$\frac{kind\ k}{k \in \Gamma_K}$$

$$\frac{type\ t :: k}{t :: k \in \Gamma_K \wedge kind(t, k)}$$

$$\forall t \in \Gamma . t :: Node \in \Gamma_K \wedge kind(t, Node)$$

3.5 Reaction Rules

One of the main extensions of bigraphs is *bigraphical reactive systems* [28], in which *pure* bigraphs are enriched with a set of *reaction rules*. These rules always follow the form:

$$reaction = redex \rightarrow reactum$$

This *reaction* definition describes a rule in which a portion of the data graph matched by the *redex* is substituted with the *reactum*. The result is an ability to express arbitrary computations, including β-reduction [25]. Consequently, bigraphical reactive systems are a powerful tool for describing both behaviour at a system level, and for expressing computation.

Fig. 3.4 provides an example of a reaction rule that might be included in a bigraphical reactive system. The rule expresses the notion that a matched A node may be replaced by a B node.

As was observed in Section 2.2, it is important to note that all reaction rules are themselves bigraphs. It is therefore conceivable that one could construct a reaction rule that operated upon other reaction rules (or indeed one that matched itself!) and modified it. This means that the Lope programming language has a *homoiconicity* property [19], i.e. a program in the language is represented using first-class values from the language itself. This gives rise to a natural kind of *metaprogramming* [15], in which programs can be constructed that operate on the structure of programs. We use this property to create an *exception* mechanism for error handling and recovery in Section 3.7.1.

The syntax for creating a reaction rule follows that of the syntax presented previously for creating nodes. We use kinds to distinguish reaction rules from other types of bigraphs:

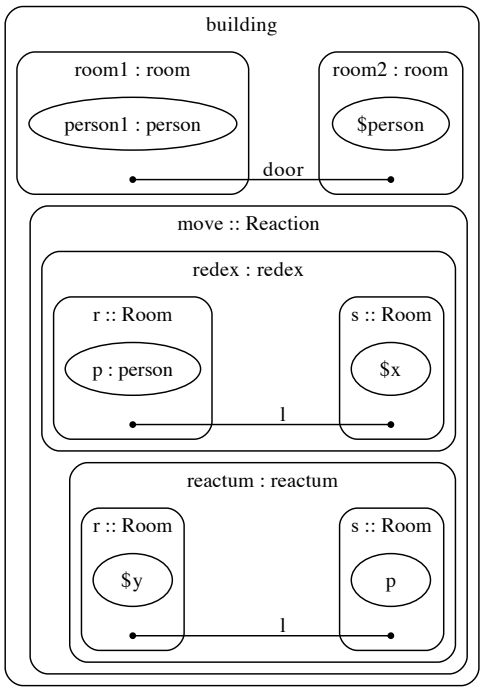
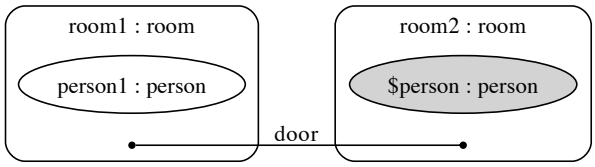


Figure 3.5: Illustration of the *building* model

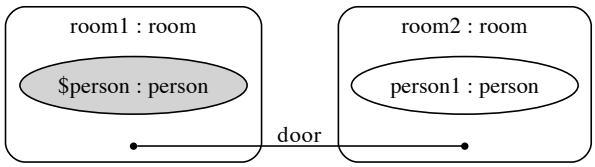
The example in Fig. 3.5 and 3.6 demonstrates several properties of reaction rules in Lope. The reaction rule *move* is within the scope of the *building* node.

```
kind Room
type person
type room :: Room

building {
    template room($person : person) : room <door> { $person }

    val person1 = person : person{}
    room1 = room(person1)
    room2 = room(_)

    link room1 <-> room2

    reaction move {
        redex {
            r <l> :: Room { p : person }

            s ($x) <l> :: Room { $x }

                        link r <-> s
        }

        reactum {
            r ($y) <l> { $y }

            s <l> { p }

            link r <-> s
        }
    }
}
```

Figure 3.6: Lope code corresponding to the bigraph given in Fig. 3.5



(a) Before rewriting



(b) After rewriting

Figure 3.7: The application of the rule *move* to the *building* bigraph

It will not be matched outside this node. The rule itself expresses the behaviour specified in Fig. 3.7, in which a node of type *person* may move from the *room* it is in into any other empty node of kind *Room* that is connected to the current *room* node. Everything matched in the redex appears in the reactum, ensuring that nothing will be deleted by the application of this rule.

3.5.1 Variable Arity Matches

Up to this point we have been explicit about the arity (i.e. number of place graph children) of the nodes we wish to match. To provide greater flexibility, we introduce a variable arity matching mechanism (the $*$ operator) and operations for adding and removing children in reactums (the $+$ and $-$ operators), and demonstrate the construction of nodes without explicit sites, permitting any number of child nodes to be contained within it. Consequently, we can create a new version of our *building* example that allows for any number of people to be contained within the rooms, using changes in the type of nodes of kind *Person* to indicate a need to move that node between rooms:

```
kind Room
kind Person

type walking_person :: Person
type sitting_person :: Person
type room :: Room
```

```
building {
    template room : room <door> {}

    room1 = room()
    room2 = room()

    room1 + { person1 : sitting_person {} }
    room2 + { person2 : walking_person {} }

    link room1 <-> room2

    reaction moveAndSit {
        redex {
            r <l> :: Room {
                p : walking_person
                r_others* :: Person
            }

            s <l> :: Room {
                s_others* :: Person
            }

            link r <-> s
        }

        reactum {
            r - { p }
            s + { q : sitting_person {} }

            link r <-> s
        }
    }
}
```

This example demonstrates the use of the node addition and subtraction operators. In contrast to the previous example, this example uses the free variable q in the reactum to introduce a new node of a different type, and the same reaction rules could be used to accommodate any number of $Person$ nodes within the system. The rule dictates that any *walking_person* should move to the adjacent room and become a *sitting_person*. We can see that this syntax provides a considerably more compact representation of complex match conditions. Because there is no node body for the r and s nodes in the reactum, it is not necessary to specify the contents of these nodes. They will

be precisely the same nodes as were originally matched, with the addition and subtraction of child nodes performed.

3.5.2 Scoping Rules

As was discussed in Section 1.6.1, previous attempts at direct modeling within bigraphical reactive systems has been dismissed as awkward because of the lack of simple recursion or ability to perform a *query* (which is just a reaction rule that matches some set of items) only within a specific context. To overcome this limitation, we propose a departure from the usual encoding of bigraphical reactive systems that places all reaction rules within the same (global) scope where they apply to all matching nodes within the system. Instead, we place reaction rules within the system itself, and confine the scope of a reaction rule to some sub-graph that is determined by its place within the place graph hierarchy. Therefore a reaction rule will only match nodes (and links) that exist within the sub-tree below its parent node in the place graph. For example:

```
type t
type u

a : t {
    b : t {
        c : t {
            d : t {}
        }

        reaction r2 {
            redex { _ : t}
            reactum {}
        }
    }

    reaction r1 {
        redex { _ : t}
        reactum {}
    }
}
```

In this example, $r1$ will match nodes b, c and d, while $r2$ will match only c and d. The parent a node will not be matched by any rule. This scoping rule

has been chosen in order to permit a restricted form of metaprogramming, as a reaction rule is always in the correct scope to match itself. Similarly, this enforces a kind of security model that permits untrusted Lope bigraphs to be inserted as a child of some trusted program, with the guarantee that it cannot affect the environment outside its root node.

One exception to this scope control mechanism is provided by the presence of links, as links are permitted to cross node boundaries. We allow a reaction rule to match any node within its scope, or *any node linked to one within its scope*. This relaxes the scope restriction somewhat, and allows for models to be constructed that permit granular access control, based on the presence of links that leave the scope of the current rule, without needing to place the rule higher up the place graph hierarchy.

We believe that this modification provides several benefits and addresses the concerns presented in [7]:

- Queries over limited contexts are achieved by selecting the appropriate location for the reaction rule within the place graph.
- A generalised form of metaprogramming becomes available that permits the implementation of other language features in an elegant manner, including exceptions (Section 3.7.1) and recursion (Section 3.7.2).
- From a software engineering perspective, we believe that limiting the scope of reaction rules makes it easier for programmers to predict the effect of adding, modifying or removing rules within the system by placing the rules closer to the data to which they are intended to apply.

3.6 Computational Sub-Language

Computation within our language is represented using an embedding of a graph rewriting system (GRS), based upon the systems defined in [5] and [34]. This graph rewriting system has the advantages of being equivalent to the λ-calculus

in expressiveness (and indeed, permits direct encoding and execution of terms in the λ-calculus) as well as fitting nicely with the bigraphical conventions introduced thus far. The graph rewrite rules used in [5] and [34] to encode familiar computational artifacts such as arithmetic, literal values, lists, recursion and conditionals can be expressed using the same mechanism introduced in Section 3.5 to express reaction rules in our agent language.

3.6.1 Ground Terms

```
0 : int :: Value, Computation {}
1 : int :: Value, Computation {}
...


nil : list :: List, Value, Computation {}


true : bool :: Value, Computation {}
false : bool :: Value, Computation {}
```

3.6.2 Arithmetic

Binary arithmetic operators are defined as nodes with two sites, relying on the fixed ordering of children to preserve the meaning of non-commutative operators:

```
+ ($a : int, $b: int) : int :: Computation {
        $a
        $b
}


- ($a : int, $b: int) : int :: Computation {
        $a
        $b
}


* ($a : int, $b: int) : int :: Computation {
        $a
        $b
}
```

```
/ ($a : int, $b: int) : int :: Computation {
        $a
        $b
}
```

3.6.3 Functions

```
apply ($body : 'a -> 'b :: Computation,
       $x : 'a :: Computation) : 'b :: Computation
{
        $body
        $x
}
```

3.6.4 Conditionals

```
if ($cond : bool :: Computation,
    $true : 'a :: Computation,
    $false : 'a :: Computation) : 'a :: Computation
{
        $cond
        $true
        $false
}
```

3.7 Metaprogramming

The introduction of metaprogramming (i.e. writing Lope programs that operate over Lope programs, including themselves) requires two basic modifications to the usual encoding of reaction rules within bigraphical reactive systems:

- The modification of scope rules described in Section 3.5.2 such that reaction rules exist as ordinary nodes within the place graph, and apply to all the children of its parent node (i.e. the rule itself and its siblings).
- Careful control of the order in which reactions are attempted to ensure a deterministic evaluation order.

The first point is demonstrated in Fig. 3.8, in which the scope of a reaction rule is always restricted to its siblings and children. The nodes at which the

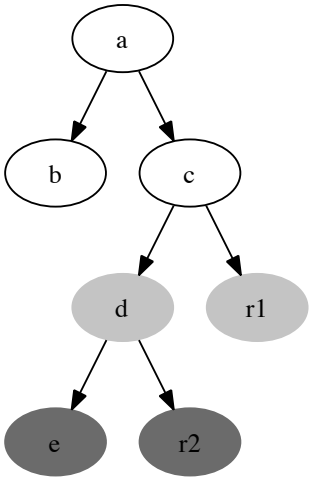


Figure 3.8: Two reaction rules with their scopes made explicit

rule could potentially be matched are shown in grey, whereas the nodes which will never be subject to matching against that rule are shown in white.

Determining the correct evaluation order proceeds similarly. The situation we wish to avoid is one where a reaction rule r_2 is within the scope of some other reaction rule r_1, where r_1 could potentially rewrite r_2 out of existence before r_2 is applied to its scope. Consequently, we define a *partial order* $\leq$ over reaction rules, based upon their location in the place graph. To define this relation, we first define the set S_n, that is the set of all nodes within the scope of some reaction rule n, using the *prnt* predicate from the place graph:

$$prnt(r, n) \wedge prnt(r, k) \rightarrow k \in S_n$$

$$prnt(v, x) \wedge v \in S_n \rightarrow x \in S_n$$

Having constructed the set S_n, we can then define the partial order relation over the set of reaction rules R:

$$x, y \in R \wedge x \neq y \wedge x \in S_y \rightarrow x < y$$

Execution then proceeds in a manner controlled by the $eval(r)$ predicate, that determines that a match of rule r has been attempted:

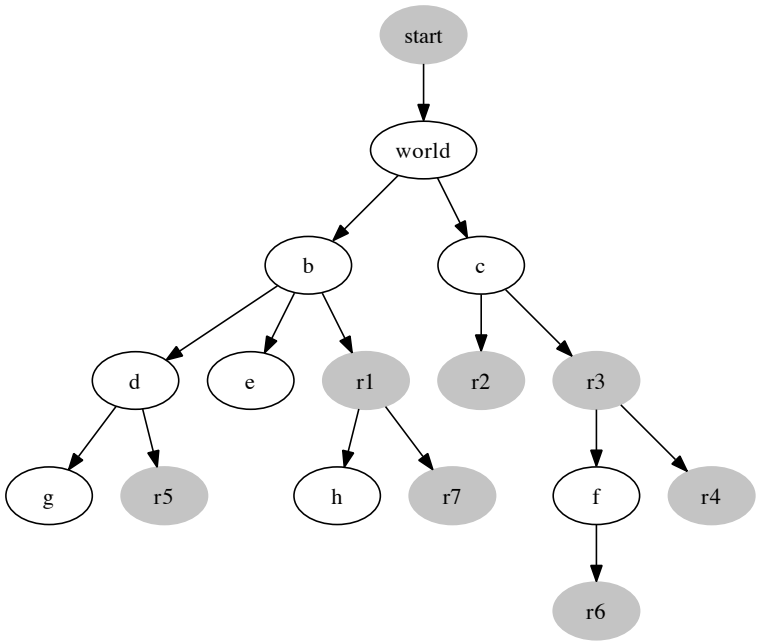
$$\forall x \in R. \forall y \in R. x < y \rightarrow eval(x); eval(y)$$

That is, for any two rules that have a strict ordering ($x < y$), x must be attempted before y. For rules without an ordering relation between them, evaluation may proceed in any order. A small exception is enforced for the computational sub-language, based upon the idea that evaluation of *Computation*-kind nodes should proceed as soon as the redex is matched. Consequently, despite them existing outside the place graph, we consider the computation rules to be strictly smaller than all other rules, such that $\forall x \in R.computation < x$. This means that standard arithmetic reductions (e.g. add, subtract, multiply, divide etc) are tested against the bigraph before the matching of any other reaction rules is attempted.

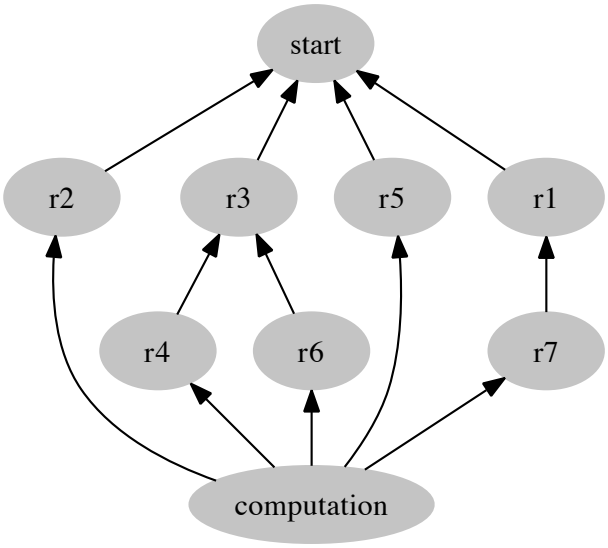
Fig. 3.9 demonstrates a system with many nested reaction rules, and the lattice that the partial order forms over the set of reaction rules (with the implied *start* rule that instantiates the global *world* node as the least upper bound and the set of computation rules, *computation* to be the greatest lower bound).

Instances where multiple reaction rules are siblings (and therefore have no order between them) are treated non-deterministically, although there exists a notion of *fairness*, such that each rule will be invoked *eventually* (provided the first rule applied does not rewrite the other one out of existence).

There is some complication with respect to the handling of scope control for links that leave the scope of the rule under consideration. We defer to the place graph in this case, even though a link could potentially connect to a node within the scope of some other reaction rule, or indeed to some node within a higher enclosing scope. Pragmatically, this appears to be uncommon, and avoiding these situations seems possible with some additional care from the programmer. We believe that a useful future extension may be to extend the ability of tools to statically detect conflicts between reaction rules and report them as errors or warnings to the programmer.



(a) The place graph



(b) The lattice over the set of reaction rules

Figure 3.9: A graph with nested reaction rules and the rule lattice

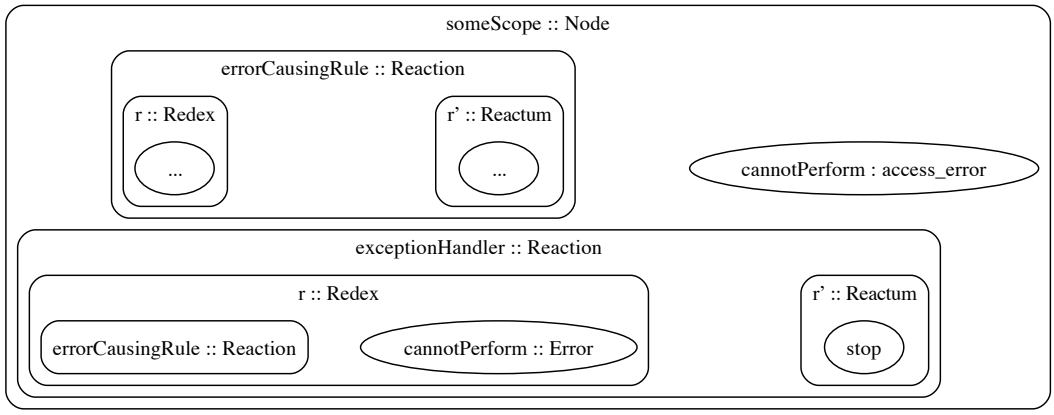


Figure 3.10: Rewriting of other reaction rules for exception handling

3.7.1 Exceptions

It is a common feature of many programs that some exceptional circumstance may occur outside the programmer's control that he or she wishes the program to handle (we provide an example of this in Section 5.2). In the distributed computing world it is important to be able to concisely describe actions such as disconnection from a network, or a service becoming unavailable (and possibly returning at some later time). We use a kind of *metaprogramming* over reaction rules to permit the encoding of these situations, and modification of the program to reflect these exceptional circumstances.

Fig. 3.10 provides an example of a reaction rule that will rewrite another reaction rule in the event that an *Error*-kind node is present in the current scope. In this case, the exception-handling reaction rule will stop any further reactions of the rule that caused the error by rewriting it to the special *stop* node. Another rule could re-establish the original rule at a later time if the underlying cause of the error changed to some other acceptable state. Error nodes such as $cannotPerform$ manifest within a Lope program when some exceptional circumstance occurs outside of the program — for example, after attempting to read a file, an *accessDenied* node might be returned instead of the contents of the file.

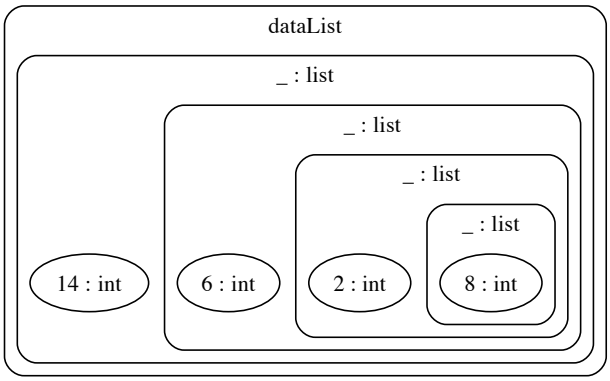


Figure 3.11: An example *dataList* bigraph

3.7.2 Recursion

One of the difficulties with direct modeling within bigraphs observed in [7] is the lack of any elegant encoding of recursion, without requiring the construction of secondary control structures (such as a call stack). We believe that the change in the encoding of reaction rules presented in Section 3.5.2 that enables metaprogramming over reaction rules provides a considerably more elegant solution to this, while still retaining a direct modeling approach.

The general encoding of recursion involves rewriting reaction rules of the form:

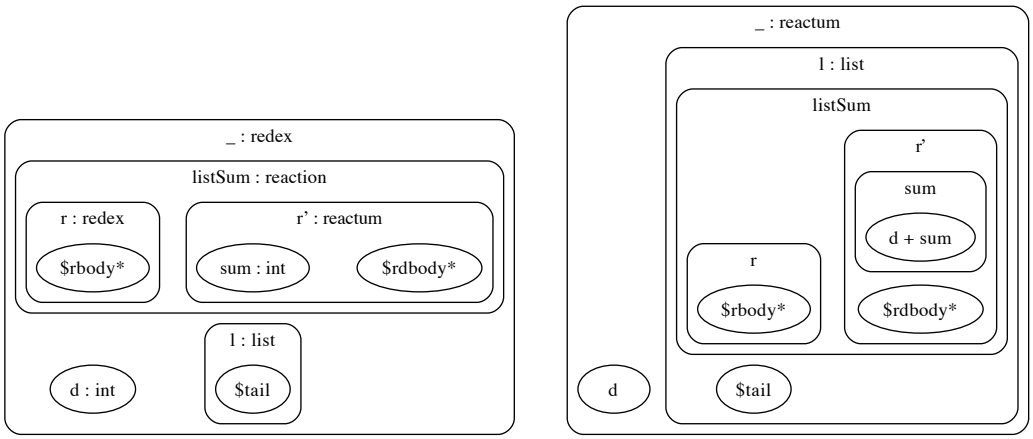
$$r \rightarrow r'$$

to the form:

$$r' \rightarrow f(r')$$

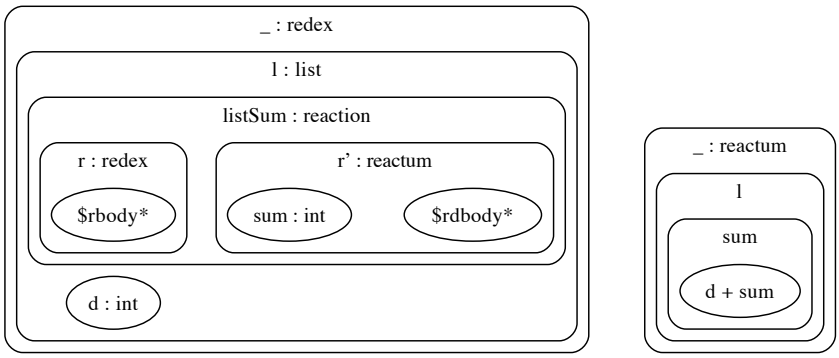
where f is some step of computation of interest. Termination is handled using the *fairness* property, such that a second reaction rule may be constructed to handle the base case of the recursion, and it is guaranteed that this rule will be attempted at least as often as the recursive case.

Fig. 3.11 demonstrates an encoding of a list data structure through the use of nested place graph nodes. Fig. 3.12 provides an example of recursion by



(a) Redex (b) Reactum

Figure 3.12: The *listSum* reaction rule



(a) Redex (b) Reactum

Figure 3.13: The *terminateSum* reaction rule

metaprogramming of reaction rules that recursively sums the elements of the list in Fig. 3.11, while leaving the original list itself unmodified. The rule moves itself lower down the place graph hierarchy with each successive application. The termination case is provided by the *terminateSum* rule (Fig. 3.13) that remains at the root, with a scope that extends to the end of the list. The only effect of this rule is to rewrite the *listSum* rule to nothing once it reaches the end of the list (i.e. a place graph node with no children), leaving the result of the summation at the end of the list.

The operation of the *listSum* is slightly counter-intuitive because the accumulation of the sum is performed within the reactum of the rule itself (by modifying the reactum of the rule as it is moved further down inside the nested list). The *sum* node in the original reactum is replaced by $sum + d$, meaning that the next application of the rule will output this modified value for *sum*, and so on until the end of the list is reached.

We believe this means of encoding recursion could be generalised to create basic traversal strategies for subgraphs to create operations such as *map* and $fold$ that operate over Lope bigraphs.

3.8 Bigraphical Agents

The basic strategy when encoding agent-style computation within bigraphs is to use the structure of the bigraph to represent the state of the agent and network environment, and to use links to express connectivity between locations. Reaction rules should be used to encode the actual computation that the agent is to perform, as well as the dynamic, network-aware behaviours.

By using this means of encoding agents serialisation of both the agent computation and accompanying data for transmission over the network is as simple as beginning serialisation from the *Process* node and recursing downwards through the place graph. The "data" that the agent carries is stored within the agent "program" (i.e. reaction rules and program structure), or is

Agent Construct	Bigraphical representation
Agent computation	Expressed using appropriate low-level reaction rules and constructs from the computational sub-language.
Network environment	Encoded with *Host*-kind nodes and links between those that mirror the real network topology.
Agent state	Stored as nodes and links inside the agent itself.
Inter-agent communication	Agents that are permitted to communicate will have links between them, permitting reaction rules that exchange sub-graphs between agents and services.

Table 3.1: An encoding of agent features as bigraphs

implied by the configuration of the agent bigraph.

Table 3.1 provides a summary of such a mapping from the constituent features of an agent to constructs within bigraphs. While other mappings may be possible, we have chosen this one as it represents a direct encoding of agent properties into bigraphs.

Chapter 4

Implementation

The prototype *Lope* system is implemented in two distinct parts: a *compiler* that translates from a high-level syntactic representation of Lope-style bigraphs into a low-level binary bytecode format, and a runtime environment that executes this bytecode representation of the program. This runtime system (akin to a *virtual machine*) provides the distributed computing features that enable computation involving multiple hosts.

Fig. 4.1 provides an overview of the phases of compilation and execution. Program text is passed through the lexing and parsing stages and a symbolic representation of the Lope bigraph is constructed internally. This representation is then used to perform code generation to the Lope bytecode format. This bytecode may then be executed by the runtime system, which interprets the bytecode to build up the graph representation in memory and begins applying reaction rules.

Table 4.1 gives the implementation technologies used for the compiler and runtime system. Standard ML was chosen as the compiler implementation

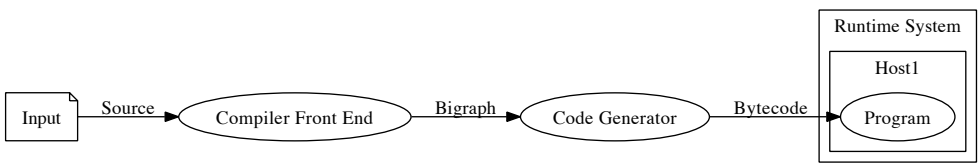


Figure 4.1: The system overview

System Component	Language	Component Output
Lexer	ML-Lex [1]	Tokens
Parser	ML-Yacc [38]	Untyped abstract syntax trees
Type Checker	Standard ML	Typed Lope Bigraph
Optimiser	Standard ML	Typed Lope Bigraph
Code Generator	Standard ML	Lope Bytecode (Appendix A)
Runtime	C	Side-effects
Services	C and others	Lope Bigraphs

Table 4.1: Implementation technologies within the Lope system

language because of the presence of a strong, static type system that prevents many of the errors commonly encountered in compiler development. The C programming language was chosen for the runtime system for ease of integration with existing third party libraries for various implementation platforms of interest (e.g. commodity PC hardware as well as specialised sensor network hardware).

4.1 Compiler Implementation

The compiler for the Lope language is fairly simple compared to most traditional compilers. Because the syntax for Lope programs (given in Section 3.3) is essentially just a means for describing the bigraphs that the Lope compiler will output, its primary function is to provide basic compile-time error checking, including enforcing the type judgements described in Section 3.4.1. The compiler provides a few basic static optimisations of the user's program, and then outputs the bigraph in a compact binary byte-code representation (described in Appendix A) suitable for execution or for transmission over network links to be executed on some remote host. Lope program text is mostly recoverable from the binary representation through mechanical reconstruction of the bigraph.

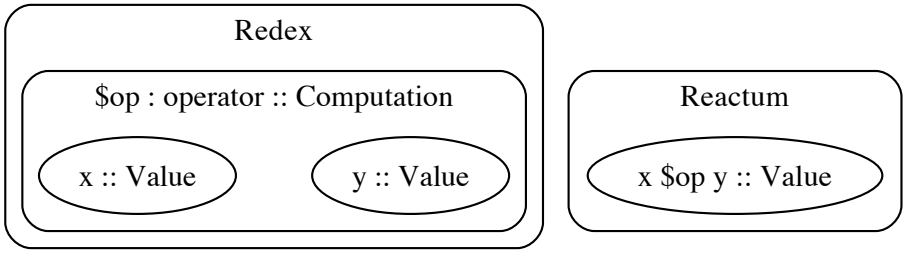


Figure 4.2: The general form for compile-time constant folding operations

4.1.1 Optimisation

A convenient side-effect of the homoiconic nature of our Lope programs is that optimisations (and indeed compilation itself) has a natural encoding as Lope reaction rules. These optimisations can therefore be implemented using the same environment used to enable runtime evaluation of reaction rules with appropriate measures taken to avoid divergent compiler behaviour. Such measures can include basic (static) checking of optimiser reaction rules to ensure that the reactum is smaller than the redex, or (in the case of our prototype implementation) an upper bound on the number of reactions that may be performed. Optimisation is "finished" when none of the optimiser reaction rules can be applied to the program bigraph.

4.1.1.1 Constant Folding

Constant folding is a compile-time optimisation performed to simplify expressions that operate on constants at compile time. For example, the value of an expression such as $1 + 2$ can be known at compile time, as the computation may be performed and replaced by a node containing the value of the result.

All constant folding optimiser reaction rules within our compiler follow the general form given in Fig. 4.2, with appropriate operators substituted in place of the variable $\$op$.

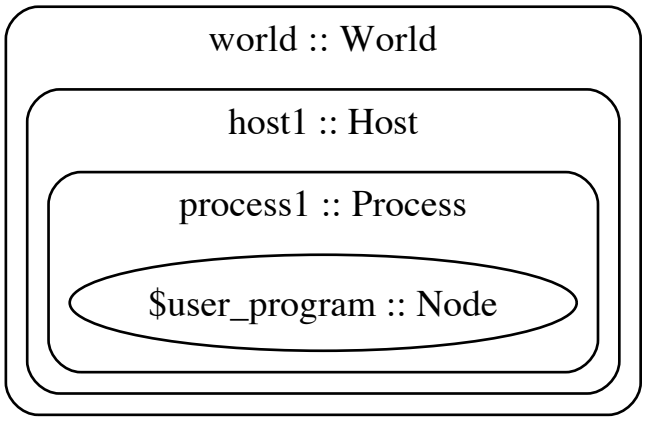


Figure 4.3: The default runtime environment *world*

4.2 Runtime Implementation

The runtime for Lope programs is responsible for interpreting the byte-code format generated by the compiler, and for exposing real-world systems as bigraphs inside the program. The default layout for the runtime environment is given in Fig. 4.3. At an application level, the Lope runtime system is similar in form and function to other virtual machines, such as the *Java Virtual Machine* [20]. However, from a programmer's perspective, it simply appears as a node container into which Lope programs may be placed.

User programs inside the runtime environment are contained within nodes of kind *Process*. The current local computer is modelled by a node of kind *Host*. As the runtime system discovers additional hosts, or loads additional processes, the bigraph that contains the user program will evolve.

In order to provide a flexible and conceptually simple language security model, the scope control mechanism for Lope reaction rules (discussed in Section 3.5.2 does not allow a user program to modify (or indeed even access) any node that exists above the level of its parent *Process* node. For well-behaved user programs this is mostly transparent, and is particularly desirable in a multi-user situation. However for certain global tasks, programs need to be able to apply outside of a limited scope. For this reason, a privileged user (i.e. an administrator) is permitted to introduce programs that operate at the

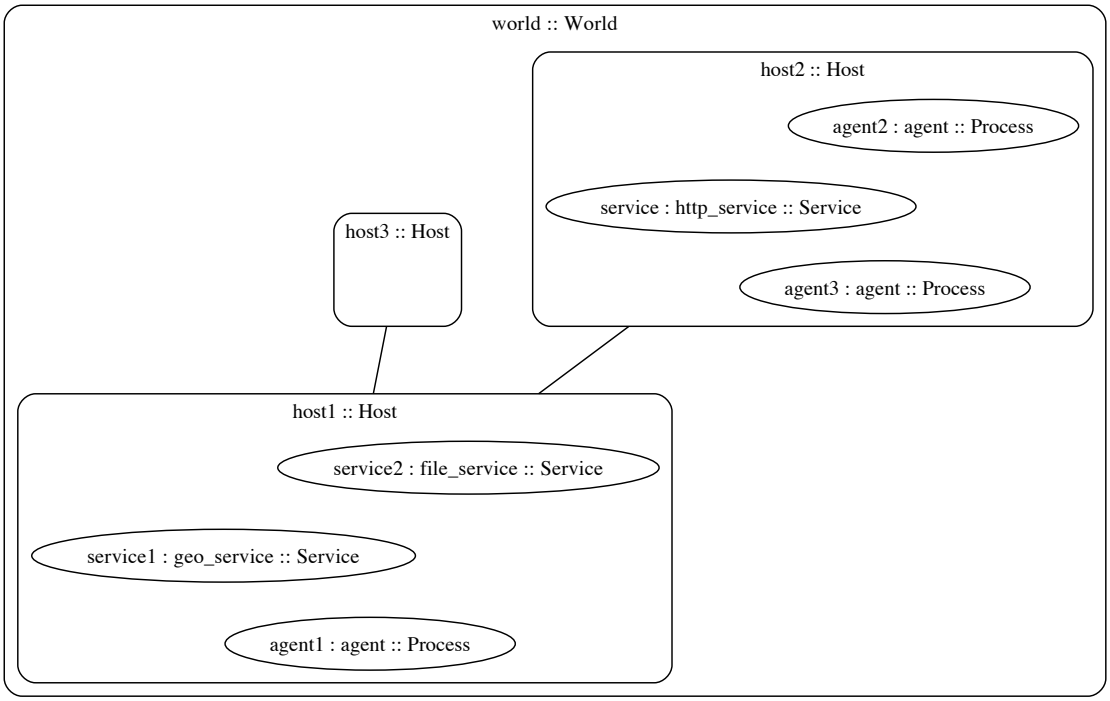


Figure 4.4: A runtime system with multiple hosts and processes

level of hosts, and indeed at the level of the *world* node. These programs can then be used to perform tasks such as software load-balancing between hosts, or automatic shutdowns (by introducing a special node of type *shutdown* into a *host* node. The same reaction rule mechanism used at the level of user programs (for modelling and computation) may be used to express high-level system behaviours, responding to the addition or removal of nodes by the runtime environment. The act of migrating a node between one host and another (through a rewrite) has the effect of physically moving that node from one computer to another. This is the basis of our implementation of mobile and agent-style processes within Lope.

Fig. 4.4 demonstrates a runtime environment consisting of many processes and hosts. The presence of multiple *Service*-kind nodes will become significant as we introduce agent-style reaction rules to this system.

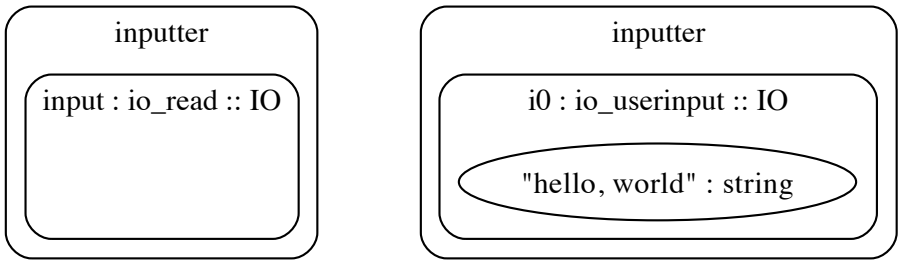
4.3 Services

Services may be defined inside or outside Lope itself; forming a part of the runtime system. A node within a particular host (of kind *Service*) may be used to permit reaction rules to be constructed that test for the presence of a certain service, however this is not a requirement in order for a service to be operating within a given runtime.

A service in the Lope sense is simply some piece of the runtime system that provides some mapping between actions and context in the real world and nodes and links within the system. For example, a *network* service might work to keep some physical network synchronised with the Lope model of that network, adding and removing hosts and links between them as they change in the real network environment. Similarly, an information service might react in the presence of a specially-crafted *query* node to perform an HTTP request to some pre-defined service, replacing the query with the result inside the runtime system.

This model extends services to a means of exposing everything that is implemented outside Lope as constructs within the Lope language. For example, a filesystem service can operate to keep a particular bigraph synchronised with an actual underlying filesystem, adding nodes as files are added to the directory, and removing files from the directory once they are deleted from the bigraph.

In implementation terms, a service operates in two ways. First, it may register a number of redexes to be matched along with all the other reaction rules within the system. In the presence of a match, the service code (implemented outside Lope) is notified with the relevant matched sub-graph as its parameter, and the matched sub-graph will be replaced by the output of the service. Second, a service running as a separate thread may send instructions to the runtime system to add or remove nodes and links from the graph. This allows a single service to both alter the bigraph in response to some external event, and to react to changes within the bigraph in order to propagate these



(a) Before service invocation (b) After service invocation

Figure 4.5: An example of an invocation of the *IO* service

Measured Event	Change in bigraph
A new host joins the network	A new *Host* node is instantiated, with a link to the node that it connected to.
A host leaves the network	All links attaching other hosts to the disconnected host are removed. Any sub-nodes of that host are removed.
A shutdown command is issued on a host	A new node of type *host_shutdown* (of kind *Control*) is instantiated within that host.
An out-of-memory event occurs	The *Process* node responsible is rewritten to the *stop* node.

Table 4.2: Mappings of events to changes in the Lope bigraph

changes to the outside world.

Fig. 4.5 demonstrates the *IO* (input/output) service within our prototype Lope implementation. This service registers a redex that ensures the service will be invoked in the presence of a node of kind *IO*. The matching node is then passed to the *IO* service (implemented in the C language in the Lope runtime), at which point it is translated into either a read operation (accepting input from the user) or as a write operation (printing the contained value to the user's screen). In the former case, the *IO* node is consumed and removed from the environment. In the latter case, the node is replaced by a node of type *io_userinput*, which is of kind *Value*.

4.3.1 Runtime System Actions

While most functionality that involves interaction with the physical world may be achieved using services, a limited set of core functionality is built into

the runtime system itself; specifically features concerned with maintaining the mapping between the physical network topology and the model of the topology represented within the Lope bigraph. Similarly, the runtime system provides bigraphical mappings of network-related tasks such as host shutdown, so as to allow the programmer to construct reaction rules to deal with these kinds of events. Table 4.2 provides the mappings between events in the physical world and their manifestation within the Lope bigraph.

4.4 Graph Rewriting

Graph rewriting involves two distinct phases. The first, *matching* involves calculating a mapping between a *redex* and nodes in the graph being matched. The second phase uses this mapping to instantiate the *reactum* and replace the matched sub-graph with the newly instantiated reactum. Nodes and links that are referenced in the redex, but not in the reactum, will be deleted by this operation. Similarly, new nodes and links may be created by referencing them only in the reactum. Updates occur when a node or link referenced in the redex occurs in the reactum with some update applied.

4.4.1 Optimisation

While the graph rewriting algorithm used within our prototype interpreter exhibits the correct behaviour, it is not particularly fast. For large graphs with many widely-scoped reaction rules, it quickly becomes inefficient. We alleviate this through the *scoping* mechanism for reaction rules (introduced in Section 3.5.2), as this limits the set of potential match sites that must be considered. Further to this, there are other optimisations that may be performed in order to improve the efficiency of the rewriting algorithm.

4.4.1.1 Arity Optimisation

By maintaining an ordered list of pairs $A : arity \rightarrow node$, we can exclude a number of potential rewriting locations. Whenever a new node is introduced as a sub-node of the $World$ node (i.e. whenever any new node is introduced to the system), its arity is recorded and the mapping $(arity(n), n)$ is added to A, sorted by arity. When considering any reaction rule that includes link graph edges within the redex, we can build a set of nodes$N$ that need to be considered:

$$N_{arity} = \{n|(a, n) \in A, a > |E_{redex}|\}$$

where $|E_{redex}|$ is the number of edges present in the redex of the reaction rule presently under consideration. This property allows us to exclude any nodes arity smaller than the number of edges we need for a successful match. We still need to consider all nodes with arity greater than $|E_{redex}|$ though, as we still allow sub-matching.

4.4.1.2 Type Optimisation

As with arity, it is possible to use the type information present on nodes to restrict the set of possible matches that must be considered. We maintain an ordered list of pairs $T : type \rightarrow node$ that records the types of any nodes present in the system. We define a total order over types, such that $t_1 \sqsubseteq t_2 \rightarrow t_1 \leq t_2$. This guarantees that any sub-type of some type t will occur later in the list of ordered pairs. Consequently, if a node in the redex of a reaction rule under consideration is annotated with a concrete type (rather than a type variable), only nodes in T greater than that type need to be considered (as this will ensure that any sub-type is also considered).

$$N_{type} = \{n|(t, n) \in T, t > redex_{root}\}$$

Kind optimisation follows from type optimisation, and can be used to re-

strict the set of potential match sites further. Where a redex specifies a kind, all nodes that have types not of that kind may be excluded from the set $_{type}$.

4.4.1.3 Combining Optimisations

With both N_{type} and N_{arity} computed, we need only to consider matching the root of the redex against nodes that appear in the set $N_{type} \cap N_{arity}$. The lists of ordered pairs need to be updated only when a reaction rule is triggered and its reactum applied, in terms of adding, modifying and removing pairs that no longer reflect the state of the system.

4.5 Inter-host communication

In implementing a runtime system that permits multiple hosts to co-exist within the same *world* node, we need some mechanism to exchange data and to provide updates across the system. Inter-host communication takes place using standard TCP sockets, with a globally-unique identifier being derived from the originating IP address (within our prototype system). This socket is then sufficient to carry a simple binary protocol, with only three operations:

- *NEWWORLD* - discard the existing host-level representation of the *world* and replace it with the serialised system state that will be sent.
- *RELOCATE process host* - serialise the process identified by *process* and send it to the host identified by *host*.
- *NOTIFYroot* - accept notification that the subgraph identified by *root* is no longer valid, and should be replaced by the serialised sub-graph that follows.

So any reaction rule that moves a node, crossing host node boundaries in the process, will be converted into a series of commands in the runtime command language. If *process*1 exists on *host*2 and a reaction rule on *host*1 dictates that the process must be relocated to *host*3, all reactions on that

process are suspended, and a $RELOCATE$ command is sent to $host3$ by $host1$. This has the effect of causing $host2$ to send a $NOTIFY$ command to $host3$, submitting the serialised process along with the command.

4.6 Security

Security is one of the fundamental challenges in any distributed system where computation may take place on computer systems not directly controlled by the user executing the program [42]. A user permitting others to execute programs on his or her computer must have confidence that a malicious or malfunctioning program cannot interfere with the operation of other programs and data present on that computer (indeed, the damage caused by computer viruses demonstrates the dangers of untrusted code being allowed to run on a computer system). Similarly, users wishing to execute programs on hosts outside their control run the risk of a malicious execution environment "spying" on their processes (perhaps capturing sensitive information or discovering the nature of the computation), or of a computation being sabotaged in such a way that it produces false or incorrect results.

While our prototype runtime system provides only a very basic model (voting) for untrusted computing, we dedicate the remainder of this section to a summary of various approaches to ensuring security within distributed systems, any of which would be applicable to our bigraphical programming language runtime system.

4.6.1 Virtualisation

The potential for untrusted code to damage the system it is running on has led to the approach generally used within *cloud computing* environments, in which each user is assigned a "virtual computer" that isolates the host computer from any potentially damaging untrusted software that is executed within the "virtual" environment. This approach does not address the issue of the user

trusting the computing resource provider (i.e. a malicious host could still spy on or interfere with programs inside the virtual environment), however more traditional notions of trust (such as business relationships and contracts) are used to limit exposure to these kinds of vulnerabilities.

4.6.2 Sandbox Security

In parasitic Javascript, the security model of the Javascript programming language is used to ensure that the potential for damage is minimised [18]. As with all Javascript code, Parasitic Javascript runs inside a kind of "sandbox" environment, in which the host executing the code can control (and easily verify) the resources and operations that a process is using. Programs are run inside the user's web browser, which actively prevents access to other processes or interference with the legitimate operation of the host computer by using a *language-based* security model, in which the language (Javascript) does not provide any features that would permit violation of some set of security constraints (e.g. no access to the filesystem, or the network). Similarly, the execution environment (the web browser, in this case) may choose to terminate a process if it exceeds some pre-determined level of CPU or memory utilisation in a way that might interfere with other processes or users on the host.

4.6.3 Direct Verification

For some computations, a result may be directly verified as a solution to a problem for some given input in a way that is computationally inexpensive (for example, the n-queens problem) [18]. These problems are well suited to untrusted computing environments, as the central controller of the computation (i.e. the host that holds the original program and input data to be distributed) may distribute a program and a problem to another host, and then verify the result against that input once it is returned, without requiring any further trust or verification that the computation was not sabotaged.

4.6.4 Voting

To prevent a malicious (or malfunctioning) host in the distributed system from sabotaging the distributed computation (by submitting false or erroneous results to their assigned computations), many volunteer-based distributing computing projects use a system of *voting*, where each computation is submitted to three or more hosts, and then majority agreement between these results is used to establish the validity of the computation. This strategy is most often used in problems where direct verification of the result is too expensive (or is computationally equivalent to simply repeating the entire computation). Unfortunately duplication of computations across nodes increases the number of hosts required to perform a single computation. In Section 4.7, we present a method by which the graph structure of our bigraphical agents may be exploited to reduce the amount of replicated computation required to implement voting.

4.6.5 User-based Security

In many distributed computing projects where users voluntarily run programs that will contribute to some global computation, a system of user authentication is often used to prevent needless verification of results returned by that user. By verifying the results submitted by newer users, a "trust score" may be calculated. Results from users that are deemed to be very trustworthy can then be accepted without necessarily requiring verification of every single result submitted. This approach is known as "spot-checking" [18].

4.6.6 Continuous Authentication

Another proposed technique that purports to prevent modification of programs on malicious hosts is a continuous authentication scheme based on *idiosyncratic signatures* [3, 12]. A piece of code that generates unique codes is hidden inside any computation that you wish to protect from modification, using code

obfuscation techniques. This code transmits these unique codes (that vary according to some cryptographic function) back to the controlling host. The controller may then verify that this stream of codes is correct with respect to the program that was submitted to the remote host. Modification of the computation should then result in the cryptographic function controlling the stream of codes changing too, which can then be detected. This does not prevent several "man-in-the-middle" attacks, however, in which a malicious program could intercept and store the stream of codes from a legitimate computation and then replay that stream of codes while performing some invalid or malicious computation.

4.6.7 Homomorphic Encryption

One of the most promising techniques that may yield a solution to both data capture and malicious modification is multiparty unconditionally secure protocols, based upon homomorphic encryption [10], which is a means of providing the ability to perform computations on encrypted data. This means that both the input and output from a given process may remain encrypted at all stages of computation, before being finally decrypted once the result is received by the controller that originally encrypted the data. As a technique, homomorphic encryption is still in its infancy and only operations such as addition, multiplication and exclusive-or can be applied to the encrypted data. While these limitations make it impractical as a complete solution to distributed computation security, it may yield useful results in the near future.

4.7 Voting Implementation

For the purposes of our prototype implementation, we assume that all hosts are trusted, however it is possible to use a special "untrusted host" type to indicate that a process distributed to this host by any reaction rule must also be distributed to n other untrusted hosts. The n untrusted copies of the process

may then be migrated back to any trusted host (of type *host*) and merged in order to verify that the processes have returned the same result. A more complete implementation might use any of the more sophisticated techniques discussed in Section 4.6 in order to perform more efficient trusted computing on untrusted hosts.

Chapter 5

Case Studies

In this chapter, we present three case studies to demonstrate the utility of our prototype Lope system, and the ability for the modified encoding of reaction rules (described in Section 3.5.2) to provide a more natural encoding of context-aware systems within bigraphical reactive systems.

5.1 Location-Aware Print Service

Following the example used in [7], we implement a model of a location-aware print service within our prototype Lope system. We assume that a user with a mobile device submits a print job to a central printer server where the job is stored. The user then travels (physically) to the nearest printer. Hypothetically, sensors are used to determine the physical location of the device that submitted the job. Once a device is in the same room as an idle printer, the job is sent to that printer and printing proceeds.

For our case study, we assume we have a single print server that can handle exactly one waiting job at any time, and three printers in three separate rooms. This initial system model is given in Fig. 5.1.

We then begin to enrich our system with agents and dynamic behaviour. We add a reaction rule *devicePrint* (Fig. 5.2) to the *building* node that moves a waiting job from a device in the building to the (empty) print spool and adds a link between the device and the job.

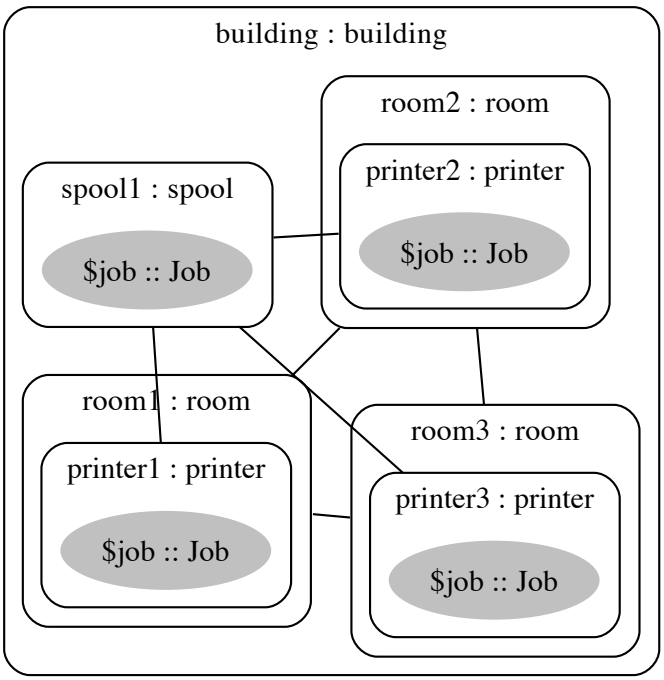


Figure 5.1: The building model for a location-aware print service

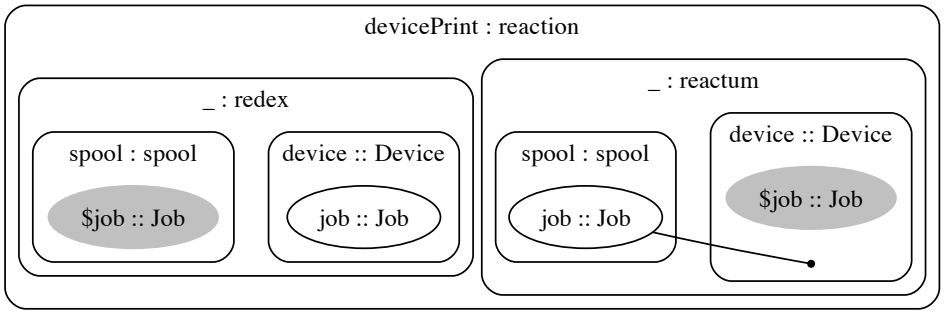


Figure 5.2: The *devicePrint* reaction rule

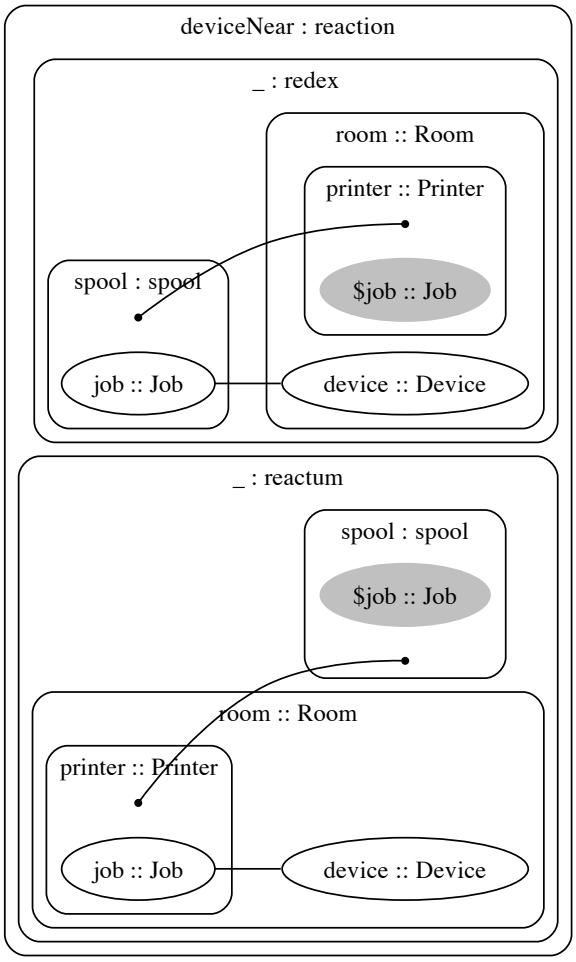


Figure 5.3: The *deviceNear* reaction rule

With the ability to send jobs to the print spool, we then enrich the system further with the rule *deviceNear* (Fig. 5.3), which moves a job from the spool to a printer once the device that submitted the job is in the same room as a printer.

Assuming that the runtime environment interface to the printer reacted whenever a job was present inside a printer node and actually performed the print job (removing the *job* node in the process), then this is all that is needed for our system to function. Lope source code and a diagram of the entire system is given in Appendix B.

5.2 Train Signalling

We demonstrate a system of trains that uses metaprogramming over reaction rules to enable exception-handling — in this case the act of some emergency alarm activating on a train.

We model our train system by a series of stations and tunnels. Exactly one train can occupy each of the two platforms in each station, and only one train may be going in each direction in a tunnel at any one time. The initial system model is illustrated in Fig. 5.4. The model makes use of the in-built *timer* node, instantiated with an *int* value and a sentinel node, that counts down until it reaches 0, at which point it resets the counter and emits the sentinel node to the current context. This allows the programmer to construct reaction rules that react to the presence of that sentinel node, which will consequently fire every *count* seconds.

We add to the system the rules presented in Fig. 5.5 that permit trains to move from station to station based on the presence of a *go* signal and a free tunnel or station platform being available in the direction of travel. The $terminatorW$ and $terminatorE$ symbols provide a turn-around for the trains (i.e. $easttrain$ trains become $westtrain$ trains and vice-versa).

Finally, we extend the system with a notion of an *emergency alarm*, that may be activated at any time. This user input is modeled by the appearance of an *emergency* node within a train. The effect of the emergency is to prevent any trains travelling in the same direction as the train with the emergency from leaving stations until the emergency is resolved. This is an ideal application of our *exceptions* mechanism described in Section 3.7.1, as we can simply rewrite the appropriate reaction rules based on the presence of the emergency signal in a train. Once the emergency symbol is replaced by the $allClear$ symbol (through user intervention), the original rules may be restored and trains may continue moving. This exception handling is illustrated in Fig. 5.6.

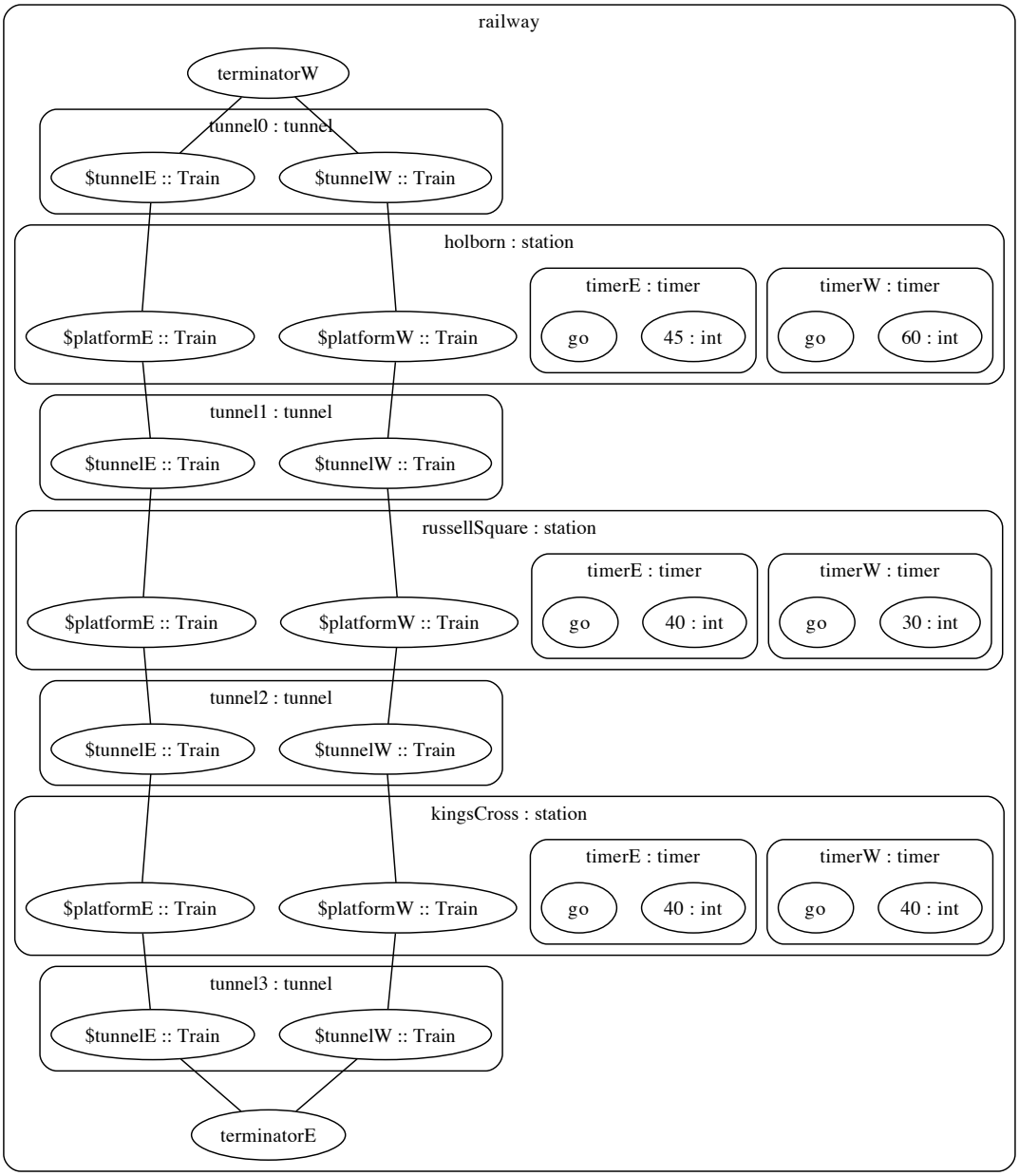


Figure 5.4: The *railway* system

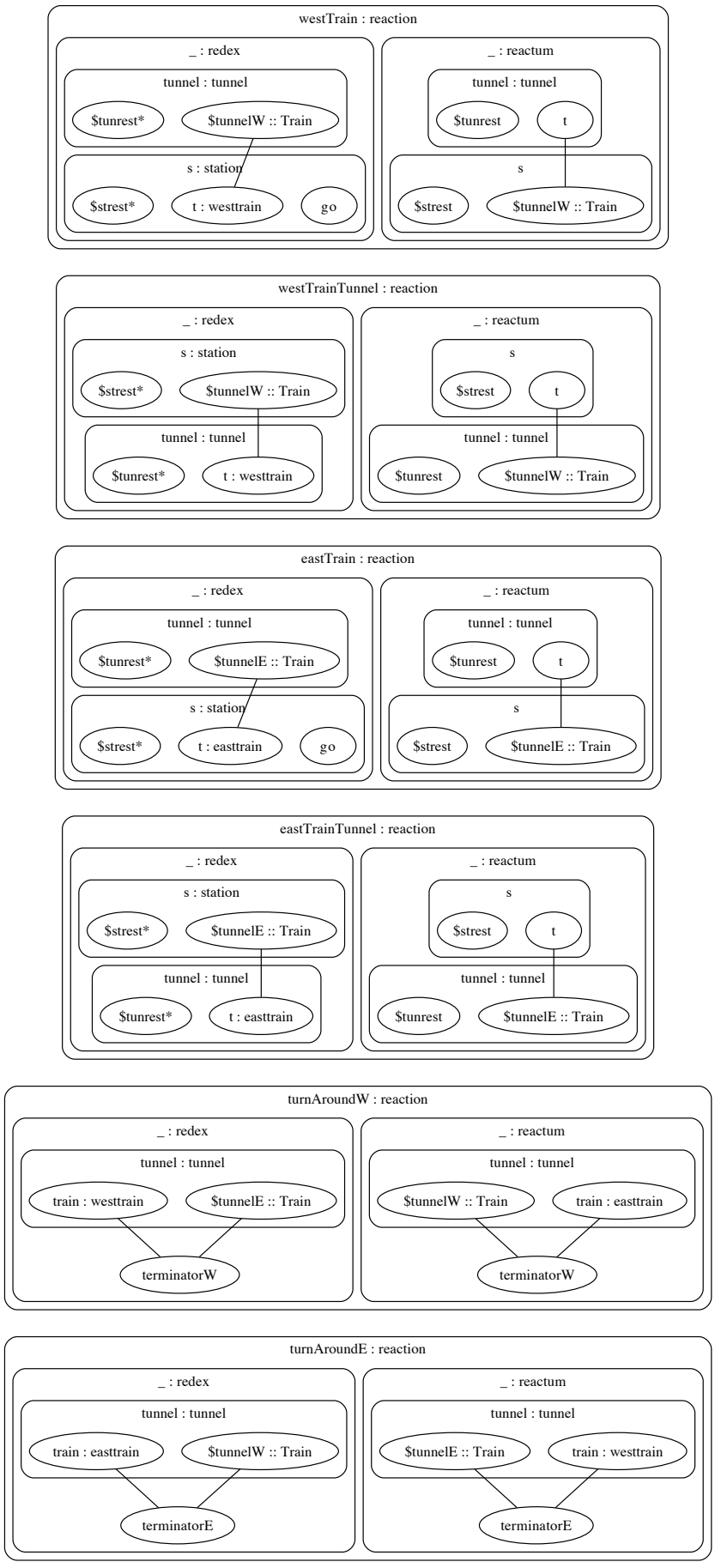


Figure 5.5: The *railway* reaction rules

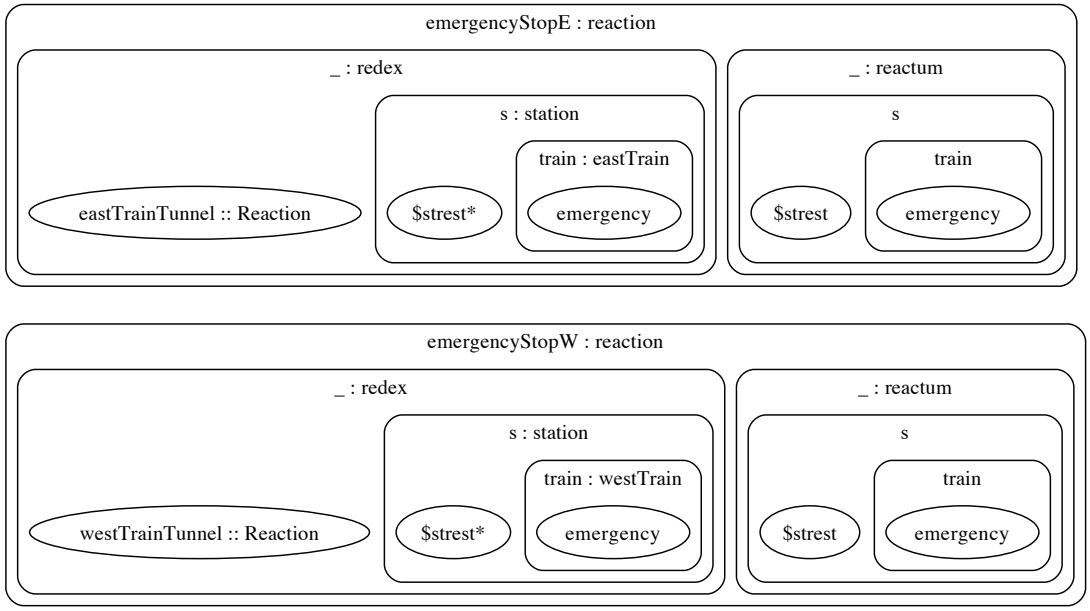


Figure 5.6: The *emergencyStop* reaction rules

5.3 Sensor Networks

There has been much recent attention upon the use of large numbers of cheap, low-power sensors to collect and collate data, possibly transmitting aggregate data over a wireless network to other nodes or to some central data collection point [35]. We believe characterising this interaction as Lope agents provides a good programming model for the reactive, asynchronous nature of this computation. Because the set of reaction rules need only to be applied when something changes in the system, the program can remain idle until a new data value is added to the Lope system bigraph, at which point rules can be tested against the new system configuration. We demonstrate a system in Fig. 5.7 that corresponds to a network of sensor nodes. The runtime system waits for an interrupt from the physical sensors, and upon a value becoming available, adds that as a node (of type *real*) to the current environment. When three such nodes are available an average is computed and transmitted back to the nearest data collection node. Fig. 5.8 shows one possible state of the system during data collection.

The *doAverage* reaction rule in Fig. 5.9 matches any three nodes of type

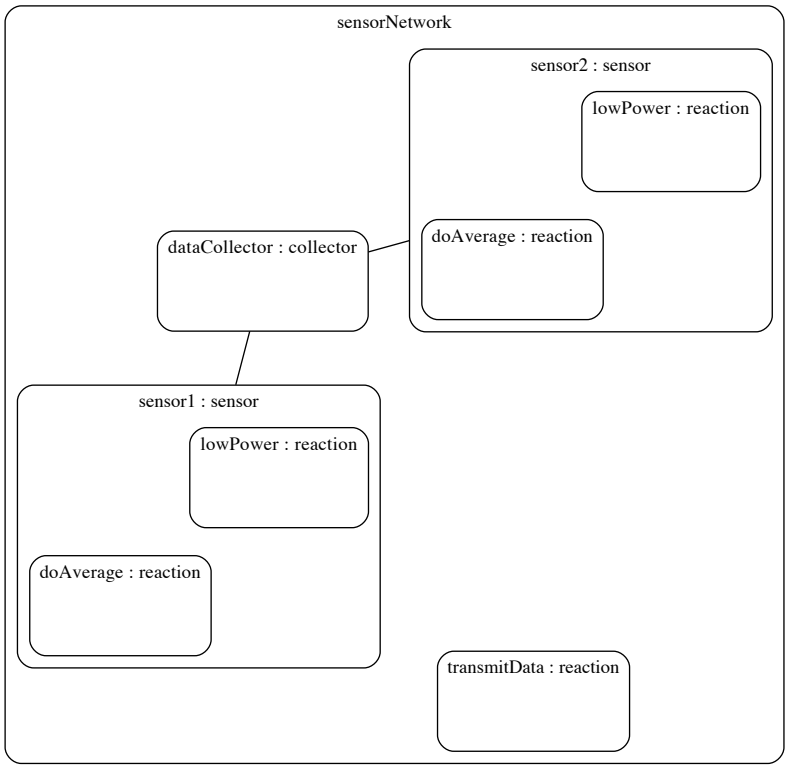


Figure 5.7: The *sensorNetwork* system after initialisation

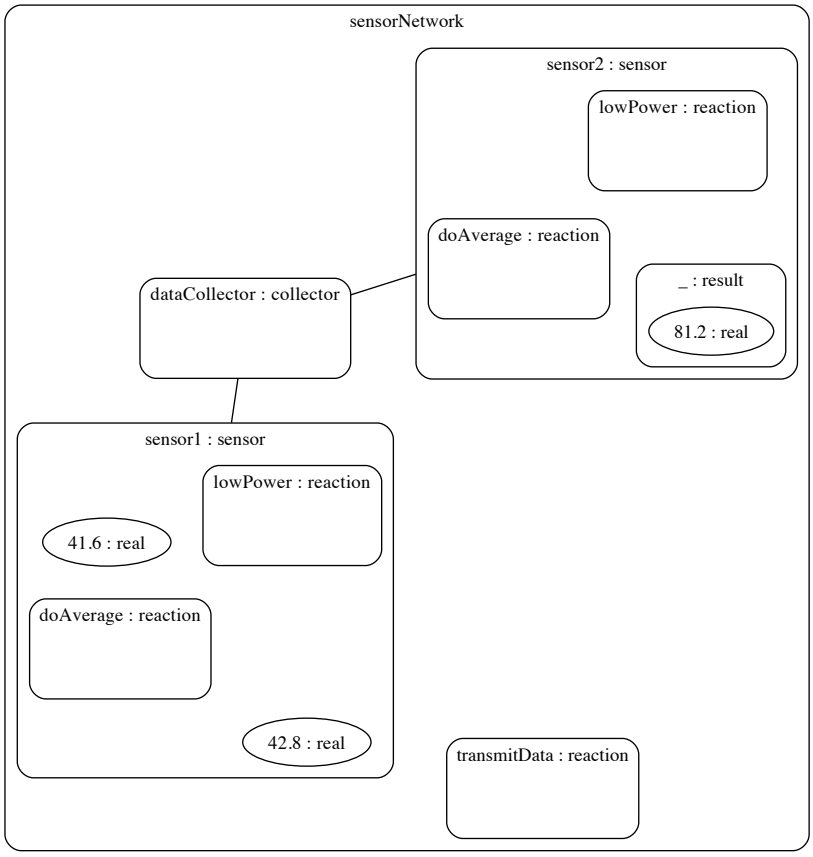


Figure 5.8: The *sensorNetwork* system during data collection

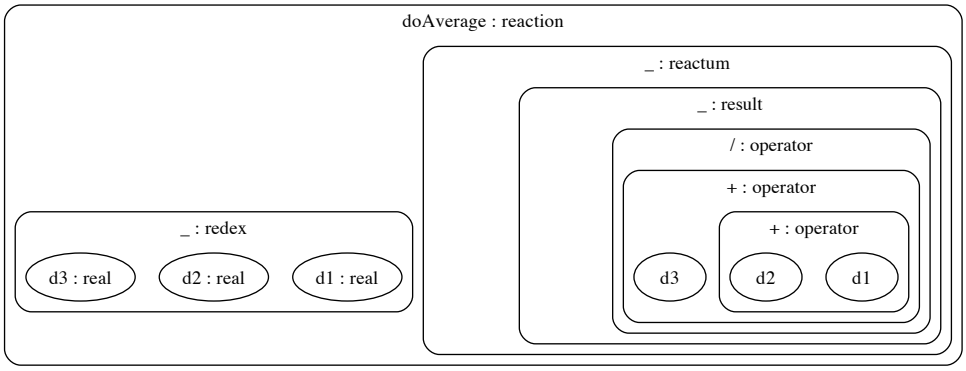


Figure 5.9: The *doAverage* reaction rule

real inside a *sensor* node (which are guaranteed to be unique by the definition of the matching process) and combines them in the reactum into a *result* node as a computation. The computation (in this case $(d1 + d2 + d3)/3$) is encoded within the computational sub-language and will be reduced to a single *real* node.

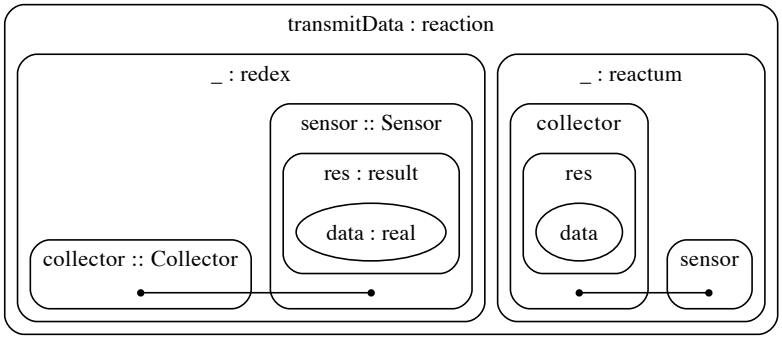


Figure 5.10: The *transmitData* reaction rule

Once the computation inside the *result* node is fully reduced the *transmitData* rule (Fig. 5.10) will match the presence of a *result* node inside a *sensor* node that is connected to a *collector*, and the result node will be copied to the collector (which then consumes it according to rules not shown within this example). The *transmitData* rule also enables a kind of *disconnected operation*, as it will only match in the presence of a link to a collector. Should this link become temporarily unavailable *result* nodes will continue to accumulate on the sensor. Once the link is restored, the rule will fire multiple times until all of the results have been copied to the collector.

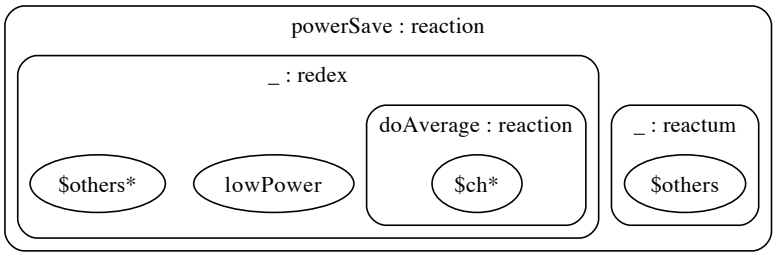


Figure 5.11: The *powerSave* reaction rule

A *lowPower* node is instantiated within the current environment by the runtime system if the node is running low on power. The desired behaviour in this case (expressed in Fig. 5.11) is to disable the computation performed on collected data so as to save power until the battery can be replaced. This is achieved using the metaprogramming techniques introduced in 3.7 to rewrite the rules involved with performing the computation. A step of manual intervention would be required to restore the reaction rule once the low power problem was remedied.

Chapter 6

Conclusion

In this thesis we have examined an approach to programming and modeling based upon metaprogramming and bigraphical reactive systems. We have detailed a prototype implementation of tools to enable the application of this approach to programming distributed agent-based systems and to ubiquitous computing. Three case studies in Chapter 5 demonstrate how this approach can be applied to encode the kinds of modeling and programming tasks that often exist in the real world — tasks that will become more common as ubiquitous computing becomes a dominant computing paradigm.

6.1 Modeling

While the Lope system described in Chapter 3 is fundamentally a *programming* language, the high-level bigraph-style constructs available to the programmer shift the primary activity from manipulation of (implicit or explicit) state through the application of functions or imperative statements to the mutation of a high-level model of the system. The case studies presented show how reaction rules can elegantly encode the dynamic behaviour of a system, either by correspondence to interactions in the real world (e.g. a train moving from one station to the next) or by the direct encoding of some rules that change some abstract system (for example, the encoding of computation). The ability to mix computation with direct, high-level modeling is one feature we believe

to be most promising about any direct modeling approach within bigraphs.

Similarly, the ability to derive visual illustrations from a program by mechanical translation further demonstrates the utility of the direct modeling approach we have proposed. Because objects within the program have a direct visual correspondence to some diagrammatic view of the real world, intuitions about dynamics and structure of a system can be reified while still retaining a "computational" flavour within the programming language.

6.2 Role of the runtime system

From the case studies presented in Chapter 5, it becomes apparent that the runtime system provides much of the low-level functionality used to provide the high-level system models with the ability to interact with the real world. Indeed, the runtime system is used to moderate the mismatch observed in [7] between the physical measured environment and the program-level model of that environment.

The runtime system (and associated services) imbues certain nodes (and types of nodes) with special meanings, and mutates the system based upon measured changes in the physical or host environment (as was presented in Section 4.2). While we have left the exact behaviour of a runtime system for a particular application and its associated services as a largely informal implementation detail, we believe there could be value in further investigation of formalising interactions between events in the real world and manifestations of these changes within the system model (and vice-versa).

As an extension of the implementation, there may be value in implementing a Lope runtime system in Javascript, so as to enable the kinds of parasitic computation described in Section 1.5 to be applied to systems expressed as Lope programs. Such an implementation would avoid the need to construct parasitic programs on a case-by-case basis—rather, an unmodified Lope program could be run in a parasitic way across a set of computers being used to

view a particular webpage that embeds the Javascript Lope runtime system.

6.3 Contributions

In pursuing a direct modeling approach to agent programming within a bigraph-style system, we were forced to confront the same issues identified in [7], as discussed in Section 1.6.1. These include the awkwardness of defining queries over limited contexts and of implementing recursion.

The solution we proposed to overcome this awkwardness was to depart from the usual encoding of a bigraphical reactive system as a bigraph and a set of globally-applicable reaction rules, and instead introduce a method of *scope control*, based on the inclusion of reaction rules themselves inside the place graph. The effect of this (presented in Section 3.5.2) was to enable the direct representation of queries over limited contexts (by placing the rule within the context of interest), and recursion (discussed in Section 3.7.2).

This in-place representation of reaction rules also has the effect of enabling *metaprogramming* over reaction rules. This ability to construct self-modifying rules provided the basis for our ability to implement recursion without the need to maintain any notion of a call-stack, as well as raising many interesting possibilities for future work to extend the power of bigraphical reactive systems as a practical modeling formalism.

The final contribution of this thesis is an exceptions mechanism for bigraphical models based upon the same metaprogramming ability we introduced within Lope. By providing the ability to rewrite reaction rules in the presence of some error state, we are afforded the ability to radically alter the behaviour of the system if we should encounter some exceptional circumstance. It seems that there is some promise in applying this exceptions mechanism to the design and construction of highly fault-tolerant distributed systems in which the behaviour of the system can degrade gracefully in the presence of hardware, software or network failures.

We hope that the application of Lope-style direct-modeling can significantly simplify the process of specifying and creating complex programs in a post-desktop model of computing. By providing an approach that is more intuitive and which successfully manages the complexity of distributed and ubiquitous systems, we believe that the software development techniques available to programmers can be bought back into alignment with the wealth of hardware and network resources that they have available to them.

References

[1] A W Appel, J S Mattson, and D Tarditi. A lexical analyzer generator for Standard ML, 1989.

[2] L Augustsson. A compiler for lazy ML. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, page 227. ACM, 1984.

[3] Mario Baldi, Yoram Ofek, Moti Yung, Torino Polytechnic, and Synchrodyne Networks. Idiosyncratic Signatures for Authenticated Execution of Management Code. *Ifip International Federation For Information Processing*, pages 204–206, 2003.

[4] A L Barabási, V W Freeh, H Jeong, and J B Brockman. Parasitic computing. *Nature*, 412(6850):894–7, August 2001.

[5] H P Barendregt, J R Kennaway, and M J Plasmeijer. Term Graph Rewriting. *East*, (87):1–37, 1986.

[6] Lars Birkedal, T C Damgaard, A J Glenstrup, and Robin (University Of Cambridge) Milner. Matching of Bigraphs. In Arend Rensink, Reiko Heckel, and Barbara König, editors, *Graph Transformation for Verification and Concurrency*, number August, pages 1—-18, 2006.

[7] Lars Birkedal, Ebbe Elsborg, Thomas Hildebrandt, and Henning Niss. Bigraphical Models of Context-aware Systems. 2005.

[8] T Brus, M van Eekelen, M O van Leer, M J Plasmeijer, and H P Barendregt. Clean-a language for functional graph rewriting. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 364–384. Springer-Verlag, 1987.

[9] Mikkel Bundgaard and Vladimiro Sassone. Typed polyadic pi-calculus in bigraphs. *Proceedings of the 8th ACM SIGPLAN symposium on Principles and practice of declarative programming - PPDP '06*, pages 1–12, 2006.

[10] David Chaum, Claude Crépeau, and Ivan Damgard. Multiparty Unconditionally Secure Protocols. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*. ACM, 1988.

[11] Ebbe Elsborg. Bigraphical Location Models, 2006.

[12] Paolo Falcarin, Riccardo Scandariato, Mario Baldi, and Yoram Ofek. Integrity checking in remote computation.

[13] A J Glenstrup. personal communication, October 21, 2009.

[14] Michael S Greenberg, Jennifer C Byington, Theophany Holding, and David G Harper. Mobile Agents and Security. *IEEE Communications Magazine*, (July):76–85, 1998.

[15] P M Hill and J Gallagher. Meta-programming in logic programming. *Handbook of Logic in Artificial Intelligence and Logic Programming: Logic programming*, page 421, 1998.

[16] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 26(1):100–106, January 1983.

[17] P Hudak, S P Jones, P Wadler, B Boutel, J Fairbairn, J Fasel, M M Guzman, K Hammond, J Hughes, T Johnsson, and Others. Report on the programming language Haskell. *ACM SigPlan notices*, 27(5):1–164, 1992.

[18] Nick Jenkin. Parasitic JavaScript (COMP520-08 Report), 2008.

[19] A Leitão. From Lisp S-expressions to Java source code. *Computer Science and Information Systems/ComSIS*, 5(2):19–38, 2008.

[20] T Lindholm and F Yellin. *Java Virtual Machine Specification*. Addison-Wesley Professional, 1999.

[21] R Milner, M Tofte, D Macqueen, and R Harper. *The definition of standard ML: revised*. The MIT Press, 1997.

[22] Robin Milner. *A calculus of communicating systems*. Springer-Verlag, New York, New York, USA, 1982.

[23] Robin Milner. *Communicating and mobile systems: the pi-calculus*. Cambridge University Press, 1999.

[24] Robin Milner. Bigraphs for Petri nets. *Computer*, pages 1–16, 2004.

[25] Robin Milner. Local bigraphs and confluence: two conjectures. In Roberto Amiado and Iain Phillips, editors, *13th International Workshop on Expressiveness in Concurrency*, number August, Bonn, 2006.

[26] Robin Milner. *The Space and Motion of Communicating Agents*. Cambridge University Press, 2009.

[27] Robin Milner and Ole Hø gh Jensen. Bigraphs and mobile processes, 2004.

[28] Robin (University Of Cambridge) Milner. Bigraphical Reactive Systems. In *Proceedings of the 12th International Conference on Concurrency Theory*, page 35. Springer-Verlag, 2001.

[29] Robin (University Of Cambridge) Milner. Pure bigraphs: Structure and dynamics. *Information and Computation*, 204(1):60–122, January 2006.

[30] David R. Morse, Stephen Armstrong, and Anind K. Dey. The What, Who, Where, When, and How of Context-Awareness. In *Conference on Human Factors in Computing Systems*, 2000.

[31] S. OConchuir. Kind bigraphs-static theory, 2005.

[32] C. A. Petri. Concurrency. *Lecture Notes in Computer Science*, 84:251—-260, 1980.

[33] Benjamin C Pierce and David N Turner. Pict: A Programming Language Based on the Pi-Calculus, 1997.

[34] Rinus Plasmeijer and Marko van Eekelen. *Functional programming and parallel graph rewriting*. Addison-Wesley, 1993.

[35] G J Pottie. Wireless sensor networks. In *Information Theory Workshop, 1998*, pages 139–140, 1998.

[36] Alexander Serenko and Brian Detlor. Intelligent agents as innovations. *Ai & Society*, 18(4):364–381, 2004.

[37] Yoav Shoham. *An Overview of Agent-Oriented Programming*, pages 271—-290. MIT Press, Cambridge, MA, USA, 1997.

[38] D Tarditi and A W Appel. ML-Yacc, 1990.

[39] D.A. Turner. *SASL language manual*. Department of Computer Science, University of Birmingham/University of Warwick, 1976.

[40] Mark Weiser. Some computer science issues in ubiquitous computing. *Communications of the ACM*, 36(7):75–84, July 1993.

[41] Lucian Wischik. New directions in implementing the pi calculus. In *CaberNet Radicals Workshop*, number August, pages 1–6, 2002.

[42] Konrad Wrona. Distributed Security: Ad Hoc Networks & Beyond. *Network Magazine*, (9):16–17, 2002.

[43] S.S. Yau, F. Karim, and S.K.S. Gupta. Reconfigurable context-sensitive middleware for pervasive computing. *IEEE Pervasive Computing*, 1(3):33–40, July 2002.

Appendix A

Lope Bytecode Format

Offset	Size	Field Name	Description
0	4	Magic	A "magic" number (44442266) that allows for identification of Lope bytecode files
4	8	Node ID	A symbol identifier for the root node
12	8	Num Children	The number of (place graph) children of the root
20	8	Match ID	A possibly-zero reference to the ID of a match for this node
28	4	Symbol Data	A possibly-zero field for data carried by the node symbol.
32	8	Type	A numerical representation of the type of the node
40	8	Kind	A kind field used in redexes where a kind is specified.
$44 * n + 4$	8	Node ID	The Node ID for the $n+1$th node in an in-order traversal
$44 * n + 12$	8	Num Children	The number of children for the $n + 1$th node.
$44 * n + 20$	8	Match ID	The Match ID for the $n + 1$th node.
$44 * n + 28$	4	Symbol Data	The Symbol Data field for the $n + 1$th node.
$44 * n + 32$	8	Type	The type field for the $n + 1$th node.
$44 * n + 40$	8	Kind	The kind field for the $n + 1$th node.
Links			
$m * 28$	8	Link ID	An identifier for the mth link entry
$m * 28 + 8$	8	Link Node	The ID of a node to connect with this link
$m * 28 + 16$	4	Link Port	The ID of the port to connect
$m * 28 + 20$	8	Link Type	The type of the link

Table A.1: The format of the bytecode, with all offsets and sizes in bytes

The bytecode format for Lope encodes bigraphs directly in a lightweight format suitable for transmission across network links and serialisation to files. Table A.1 provides a list of offsets and field sizes (given in bytes) within a serialised Lope program, based upon the number of nodes (ranged over by n) and the number of link connection points (ranged over by m). Values are expected to be represented using *little endian* word order.

Appendix B

Location-Aware Print Service Source Code

```
kind Job
kind Room
kind Printer
kind Device

type building
type room :: Room
type printer :: Printer
type spool
type mobilephone :: Device

building : building {
        spool1 <pr> : spool {
                $job :: Job
        }

        template room($printer :: Printer) <door> : room {
                $printer
                link printer <-> spool1
        }

        room1 = room(printer <pr> : printer { $job :: Job })
        room2 = room(printer <pr> : printer { $job :: Job })
        room3 = room(printer <pr> : printer { $job :: Job })

        link room1 <-> room2 <-> room3

        reaction devicePrint {
                redex {
                        spool : spool {
                                $job :: Job
                        }

                        device :: Device {
                                job :: Job
                        }
                }

                reactum {
                        spool {
                                job
                        }

                        device <owner> {
                                $job :: Job
                        }

                        link job <-> device
```

```
                }
        }

        reaction deviceNear {
                redex {
                        spool <pr> : spool {
                                job <owner> :: Job
                        }

                        room :: Room {
                                printer <pr> :: Printer {
                                        $job :: Job
                                }

                                device <owner> :: Device

                                link printer <-> spool
                                link device <-> job
                        }
                }

                reactum {
                        spool <pr> {
                                $job :: Job
                        }

                        room {
                                printer <pr> {
                                        job <owner>
                                }

                                device <owner>

                                link printer <-> spool
                                link device <-> job
                        }
                }
        }
}
```