Strategic Modelling with Graph Rewriting Tools

Olivier Namet

Doctorate of Philosophy in Computer Research
King's College London
Department of Informatics

Abstract

To model complex systems, graphical formalisms have clear advantages: they are more intuitive and make it easier to visualise a system and convey intuitions or ideas about it. Graph rewriting rules can be used to model their dynamic evolution and from a practical point of view, graph transformations have many applications in specification, programming, and simulation tools.

Strategic rewriting has been studied for *term* rewriting systems, and there are languages that allow the user to specify a strategy controlling the use of rewrite rules and to apply it. For graph rewriting, some graph-transformation languages and tools allow the users to specify the way rules are applied. However, there is no general language that uses positions explicitly to apply rules.

The work presented in this thesis describes a new notion of *located graphs* and a strategy language containing *focusing* constructs. In a graph, there is no notion of a root so standard term rewriting strategies based on top-down or bottom-up traversals do not make sense in this setting. We solve the problem by introducing the notion of position and banning in our located graphs and strategy language to allow for graph traversals and selective rule application based on location within a graph.

Two tools graphPaper and PORGY are also described, which allow users to create graph rewriting systems and to apply strategies on them to create trace trees forming result sets. Specifically, the full implementation of graphPaper, a tool to create and edit graph rewriting systems, is described as well as the implementation of the strategy language into the PORGY system to allow users to view a dynamic trace of the computation of a strategic graph system.

Acknowledgements

I would like to thank first and foremost my supervisor, Maribel Fernández, for her constant support and her critical (but always constructive) comments on my work. It has been a great pleasure and honour working with her for the last four years and a definite highlight of my PhD. I will greatly miss our theoretical discussions as well as our shared love of dance!

I am also grateful to Ian Mackie who supervised my Master's project and is responsible for introducing me to the world of graph rewriting.

I would like to thank Sebastian Hunt and Detlef Plump for accepting to be examiners for my thesis.

I would also like to thank the PORGY team: Oana Andrei, Hélène Kirchner, Guy Melançon and Bruno Pinaud for the invaluable experience of doing both practical and theoretical work in team. Their experience and knowledge have been an important contribution to my learning experience.

I would like to thank my friends Sami Ali Adib, Katie Hart, Daniel Lawrence, Holly Russell-Allison, Nicholas Swart and Antonio Zardis (to only name a few) for their support in times of stress and particularly Katie for feeding my cats when I was away at conferences!

I cannot begin to thank my mother, Marie-Françoise Namet, enough for all the support a mother gives and for always believing in me even when I doubted myself.

Finally, I would like to dedicate this thesis to the memory of my father, Alain Namet.

Contents

List of Figures

Chapter 1

Introduction

Graphs as a Modelling Tool and Graph Rewriting

To model complex systems, graphical formalisms have clear advantages, in particular in the earlier phases of the specification: graphical formalisms are more intuitive to represent data and relations and make it easier to visualise a system and convey intuitions or ideas about it.

A lot of models can be represented as graphs: networks of servers, natural language processing, chemical structures, biological entities (such as proteins and DNA), maps (for geographical regions and for transport infrastructures ranging from roads to trains) and mathematical applications for fractals and geometry.

Some graph formalisms use a textual language to describe and view graphs. This seems counterproductive to the natural visual aspect of graphs and therefore this thesis will seek to prioritise visual methods to create and view graphs. Tools such as TULIP[12, 14, 13] and ASK-GraphView[4] allow users to view graphs of any size and allow data to be extracted from graphs or even highlighted directly on a visual representation of the graph.

Graphs on their own represent a static modelling tool, showing us only a snapshot of the model. Using rewriting on graphs would allow the graph to evolve and provide more depth to the modelling. Rewriting [15, 72] is a transformation process based on the use of rules that are applied to syntactic objects such as words, terms, programs, proofs and more importantly *graphs*. It has a variety of applications; for instance, it is used to simplify algebraic expressions in computer algebra, to perform syntactic analysis of programs or natural language expressions, to define the operational semantics of a programming language, to study the structure of a group or a monoid, or to express the computational content of a mathematical proof. Other practical applications include program refactoring, static analysis, code optimisation in compilers, specification of security policies, or the modelling of complex systems in biology[9, 20, 27, 32, 22, 31].

Combined with graph rewriting rules, graphs can show a visual and dynamic evolution of a model, for instance a series of chemical reactions.

Computing by graph rewriting is also a fundamental concept in concurrency. From a theoretical point of view, graph rewriting comes with solid logic, algebraic and categorical foundations [30, 35], and from a practical point of view, graph transformations have many applications in specification, programming, and simulation tools [35]. In this thesis, we use *port graph rewriting systems* (a

general class of graph rewriting systems that have been successfully used to model biochemical systems, interaction net systems and networked servers) for the development of a visual modelling tool.

Rewriting enhances modelling with graphs by making the graphs dynamic. Applying a rule on a graph can produce more than one result (non-determinism) if the rule can be applied in more than one location in the graph, creating a tree of derivations also known as a *trace tree*. Knowing this, a control method is required to be able to guide the rewriting and therefore the evolution of the model. For this, in this thesis we design a strategy language for port graph rewriting.

Strategy in the Context of Rewriting

Reduction strategies define which (sub)expression(s) should be selected for evaluation and which rule(s) should be applied (see [50, 25] for general definitions). These choices affect fundamental properties of computations such as laziness, strictness, completeness, termination and efficiency, to name a few (see, e.g.,[76, 73, 54]). Used for a long time in λ-calculus [18], strategies are present in programming languages such as Clean [64], Curry [48], and Haskell [49] and can be explicitely defined to rewrite terms in languages such as ELAN [24], Stratego [75], Maude [59] or Tom [17]. They are also present in graph transformation tools such as PROGRES [69], AGG [36], Fujaba [61], GROOVE [68], GrGen [44] and GP [67]. The strategy language defined in Chapter 5 draws inspiration from these previous works, but a distinctive feature of the language is that it allows users to define strategies including not only operators to combine graph rewriting rules but also operators to define (and change throughout the execution) the location in the target graph where rules should, or should not, apply.

The language offers a set of primitives to select rewriting rules and a set of primitives to select the *position* where the rules apply. Alternatively, positions could be encoded in the rewrite rules using markers or conditions (as done in other languages based on graph rewriting which do not have focusing primitives). We prefer to separate the two notions (positions and rules) to make programs more readable (the rewrite rules are not cluttered with encodings), and easier to maintain and adapt. In this sense, the language facilitates separation of concerns: for example, to change the traversal algorithm it is sufficient to change the strategy and not the whole rewriting system including the rules.

The graphPaper & the PORGY Tools

This thesis presents two tools used for strategic modelling using graph rewriting. graphPaper is a visually intuitive graph editor that allows users to create, edit and visualise port graphs and port graph rewriting rules. Users can create graphs and rules by drawing only two shapes with a mouse and graphPaper will use context to determine the user's intention (for example drawing a line between two ports connects them whereas drawing a line across an edge deletes the edge). In this way, graphPaper aims to allow users to create graphs and rules as if they were using a pen and paper with the added dynamic nature that computers provide . The PORGY tool, designed

by an INRIA associate team consisting of Maribel Fernández[1] and Olivier Namet[2] (King's College London), Hélène Kirchner[3] (INRIA), Bruno Pinaud[4] and Guy Melançon[5] (INRIA and Université de Bordeaux) and Oana Andrei[6] (University of Glasgow) allows the creation and editing of graph rewriting systems as well but also lets the user apply rules and strategies (written in the language defined in Chapter 5) onto graphs and to visualise the evolution of the model in the form of a *trace tree*. PORGY also contains debugging and data analysis features to help users construct and view their models.

Main Contributions:

The main contribution of this thesis is a language for strategic modelling of complex graph rewriting systems, using *focusing* constructs to control the location of the rewriting. The strategy language allows users to describe not only the rules that need to be applied, but also the location where they apply, and the propagation mechanism controlling successive applications of rules using *focusing* constructs. The latter is complicated by the fact that in a graph there is no notion of a root, so standard term rewriting strategies based on top-down or bottom-up traversals do not make sense in this setting. The strategy language provides a distinct set of constructs that allow users to build subgraphs of the current graph they are rewriting on and to then set these subgraphs as the *position* and *banned* subgraphs of the graph (where rewrite rules must at least overlap with the *position* subgraph and cannot have anything in common with the *banned* subgraph).

A selection of different applications are presented as case studies with fields ranging from games and AI to arithmetic and geometric (fractal generation) modelling.

Also, the implementation of graphPaper, a tool to create and edit graph rewriting systems, is described as well as the design and implementation of PORGY, a visual and interactive tool developed to model complex systems using port graphs and port graph rewrite rules guided by strategies, and to navigate in the derivation history.

The results in Chapter 5 were developed in collaboration with Oana Andrei, Maribel Fernández, Hélène Kirchner, and Bruno Pinaud and the chapter expands and improves the results described in the papers listed in the Publication section. As an author of these papers, I contributed fully to the research, the writing up, and was in charge of presenting the papers at all of their respective conferences or workshops.

The examples described in Chapter 6 were all created by the author of this thesis and some (or parts) of these examples were included in published papers (See Publication section) and demonstrated at various conferences, workshops and classes.

The results in Chapter 7 expand on the work of the INRIA associate team (mentioned previously) where the author's main contributions to PORGY are the Strategy Engine and making design decisions including the GUI and interface of PORGY.

[1] http://www.dcs.kcl.ac.uk/staff/maribel/
[2] http://www.oliviernamet.co.uk
[3] https://wiki.bordeaux.inria.fr/Helene-Kirchner/doku.php
[4] http://www.labri.fr/perso/bpinaud/
[5] http://www.labri.fr/perso/melancon/
[6] http://dcs.gla.ac.uk/ oandrei/

Publications:

- **Graph Creation, Visualisation and Transformation**[39] (RULE 2009 - Maribel Fernández, Olivier Namet)

  Abstract: We describe a tool to create, edit, visualise and compute with interaction nets - a form of graph rewriting systems. The editor, called GraphPaper, allows users to create and edit graphs and their transformation rules using an intuitive user interface. The editor uses the functionalities of the TULIP system, which gives us access to a wealth of visualisation algorithms. Interaction nets are not only a formalism for the specification of graphs, but also a rewrite-based computation model. We discuss graph rewriting strategies and a language to express them in order to perform strategic interaction net rewriting.

- **Strategic programming on graph rewriting systems**[40] (IWS 2010 - Maribel Fernández, Olivier Namet)

  Abstract: We describe a strategy language to control the application of graph rewriting rules, and show how this language can be used to write high-level declarative programs in several application areas. This language is part of a graph-based programming tool built within the port-graph transformation and visualisation environment PORGY.

- **PORGY: Strategy-Driven Interactive Transformation of Graphs**[8] (TERMGRAPH 2011 - Oana Andrei, Maribel Fernández, Hélène Kirchner, Guy Melançon, Olivier Namet, Bruno Pinaud)

  Abstract: This paper investigates the use of graph rewriting systems as a modelling tool, and advocates the embedding of such systems in an interactive environment. One important application domain is the modelling of biochemical systems, where states are represented by port graphs and the dynamics is driven by rules and strategies. A graph rewriting tool's capability to interactively explore the features of the rewriting system provides useful insights into possible behaviours of the model and its properties. We describe PORGY, a visual and interactive tool we have developed to model complex systems using port graphs and port graph rewrite rules guided by strategies, and to navigate in the derivation history. We demonstrate via examples some functionalities provided by PORGY.

- **A Strategy Language for Graph Rewriting** (LOPSTR 2011 - Maribel Fernández, Hélène Kirchner, Olivier Namet)

  Abstract: We give a formal semantics for a graph-based programming language, where a program consists of a collection of graph rewriting rules, a user-defined strategy to control the application of rules, and an initial graph to be rewritten. The traditional operators found in strategy languages for term rewriting have been adapted to deal with the more general setting of graph rewriting, and some new constructs have been included in the language to deal with graph traversal and management of rewriting positions in the graph. This language is part of the graph transformation and visualisation environment PORGY.

Outline of the Rest of the Thesis

- **Chapter 2 Background** motivates the use of port graph rewriting systems and interaction nets by providing both formalisms as well as some of their properties.
- **Chapter 3 Related Work**
  - **Section 3.1 Graph and Graph Rewriting** describes different graph and graph rewriting systems and motivates our choice of port graphs.
  - **Section 3.2 Graph Rewriting Tools** describes different tools that use graph rewriting and motivates the design decisions behind PORGY.
  - **Section 3.3 Strategic Term & Graph Rewriting Tools** describes different kinds of existing strategic modelling and explains the importance of the notion of position in strategic graph rewriting.
- **Chapter 4 graphPaper : A Tool to Create Graph Rewriting Systems** gives us an overview of the idea behind graphPaper's unique visual and interactive interface as well as a description of the implementation.
- **Chapter 5 Strategy Language**
  - **Section 5.1 Introduction** presents certain new concepts such as *located graphs*, *graph programs* and *result sets*, and the notion of *termination*.
  - **Section 5.2 Syntax** defines the grammar of the strategy language in 4 categories: *focusing*, *transformations*, *applications* and *strategies*.
  - **Section 5.3 Semantics** shows and explains the semantic rules for each operator in the language used when executing a strategy.
  - **Section 5.4 Properties** explores and proves several interesting properties of the strategy language: *termination*, *characterisation of normal forms*, *result set* and *completeness*.
- **Chapter 6 Application of the Strategy Language**
  - **Section 6.1 Non-Primitive Operators** shows some non-primitive operators that make the language easier to use such as `not()`, `try()` and `repeat()`.
  - **Section 6.2 Traversals** explains how several common types of traversals (outermost, innermost and interface normal form) are achieved with the strategy language.
  - **Section 6.3-6.9 Various Examples** describes different models using port graphs and the strategy language: arithmetic using interaction nets, Von Koch fractals, Sierpinski's Triangle generation, AI in a game of pacman, finding the shortest path in a labyrinth, flag sorting and biomolecular reaction in the AKAP model.
- **Chapter 7 PORGY** details the motivation behind the PORGY tool, its features and how the backend works. The latter uses TULIP to provide a powerful data structure (allowing for

very large graphs) and a strong visual interface using OpenGL. The Chapter also describes the general interface of the tool and all the plugins created to make PORGY including the pattern matching algorithm and the strategy engine.

- **Chapter 8 Conclusion & Future Work** concludes the thesis by summarising the thesis and describing the contributions presented in this thesis as well as future work for the strategy language and the graphPaper and PORGY tools.

Throughout the thesis the shorthand notation Section $X.Y$ is used to mean Chapter X, Section Y.

Chapter 2

Background

2.1 Graph Rewriting Systems

There are several definitions of graph rewriting, using different kinds of graphs and rewriting rules (see, for instance, [29, 46, 19, 66, 21, 52]). For this thesis, we consider the port graph rewriting systems introduced in [6], of which interaction nets [52] can be seen as a particular case. The main originality of port graphs is the presence of labelled ports on nodes, and attributes and values that can be associated to nodes and ports. An edge between two nodes is actually connected through a port of each node, and attributes and variables can be used to store extra data and change the way rewriting happens. This system allows for a more natural modelling of systems such as biochemical reactions (where for example a protein can be seen as a node, its different connection sites as ports and attributes the state of each site and of the actual protein).

We recall below the main notions of port graph rewriting and interaction nets.

2.1.1 Port Graphs

Intuitively, a port graph is a graph where nodes have explicit connection points called *ports*; edges are attached to ports.

Definition 1. *We define a* p-signature ∇ *with:*

- $\nabla_{\mathscr{N}}$*, a set of node labels.*
- $\mathcal{X}_{\mathscr{N}}$*, a set of node label variables.*
- $\nabla_{\mathscr{P}}$*, a set of port labels*
- $\mathcal{X}_{\mathscr{P}}$*, a set of port label variables.*
- $\nabla_{\mathscr{A}}$*, a set of attribute labels.*
- $\nabla_{\mathscr{V}}$*, a set of value labels.*
- $*$*, representing a variable element to be used with* $\nabla_{\mathscr{A}}$ *and* $\nabla_{\mathscr{V}}$ *when a specific label of a value and attribute is not important.*

We assume that $\nabla_{\mathcal{N}}$, $\mathcal{X}_{\mathcal{N}}$, $\nabla_{\mathscr{P}}$, $\mathcal{X}_{\mathscr{P}}$, $\nabla_{\mathscr{A}}$ *and* $\nabla_{\mathscr{V}}$ *are pairwise disjoint, to avoid confusion.*

Each node label N *has a finite set of port labels* $Interface(N) \subseteq \nabla_{\mathscr{P}}$ *and* $Interface(X) \subseteq \nabla_{\mathscr{P}} \cup \mathcal{X}_{\mathscr{P}}$ *for all* $X \in \mathcal{X}_{\mathcal{N}}$.

Similarly, each node label N *has a finite set of attribute labels* $Attribute(N) \subseteq \nabla_{\mathscr{A}}$ *and each port label* P *(such that* $P \in Interface(N)$*) has a finite set of attribute labels* $Attribute(N, P) \subseteq \nabla_{\mathscr{A}}$.

A labelled port graph *over a* p-signature ∇ *is a tuple* $G = (V_G, lv_G, E_G)$ *where:*

- V_G *is a finite set of nodes.*
- lv_G *is a function that returns, for a given node* $n \in V_G$*, a node label (the node's name), a set of port labels (each with their own sets of attribute labels and assigned value labels), and a set of attribute labels (each with a value label). This can be represented using a record:*

$$\begin{aligned} lv_g(n) \quad = \quad & [name, \\ & (p_1, a_1^1) = V_{1,1}, (p_1, a_1^2) = V_{1,2}, \ldots, (p_n, a_n^m) = V_{n,m}, \\ & a_0^1 = V_{0,1}, \ldots, a_0^n = V_{0,n}] \end{aligned}$$

$$\begin{aligned} \textit{where:} \qquad n \quad & \in \quad V_G \\ name \quad & \in \quad \nabla_{\mathcal{N}} \cup \mathcal{X}_{\mathcal{N}} \\ V_{n.m} \quad & \in \quad \nabla_{\mathscr{V}} \cup \{*\} \\ p_n \quad & \in \quad \nabla_{\mathscr{P}} \cup \mathcal{X}_{\mathscr{P}} \\ a_n^m \quad & \in \quad \nabla_{\mathscr{A}} \end{aligned}$$

lv_G can also be seen as a family of functions:

$lv_G name:$	$V_G \to \nabla_{\mathcal{N}} \cup \mathcal{X}_{\mathcal{N}}$ $lv_G name(n) = N$
$lv_G ports:$	$V_G \to \nabla_{\mathscr{P}} \cup \mathcal{X}_{\mathscr{P}}$ $lv_G ports(n) = Interface(lv_G name(n))$
$lv_G portAttributes:$	$V_G \times \nabla_{\mathscr{P}} \to \nabla_{\mathscr{A}}$ $lv_G portAttributes(n)(P) = Attribute(lv_G name(n), P)$ $\quad$ *where* $P \in Interface(lv_G name(n))$
$lv_G portAttributeValue:$	$V_G \times \nabla_{\mathscr{P}} \times \nabla_{\mathscr{A}} \to \nabla_{\mathscr{V}}$ $lv_G portAttributeValue(n)(P)(A) = V$ $\quad$ *where* $P \in lv_G ports(n)$ $\quad\quad A \in lv_G portAttributes(n)(P)$ $\quad\quad V \in \nabla_{\mathscr{V}} \cup \{*\}$
$lv_G attributes:$	$V_G \to \nabla_{\mathscr{A}}$ $lv_G attributes(n) = Attribute(lv_G name(n))$
$lv_G attributeValue:$	$V_G \times \nabla_{\mathscr{A}} \to \nabla_{\mathscr{V}}$ $lv_G attributeValue(n)(A) = V$ $\quad$ *where* $A \in lv_G attributes(n)$ $\quad\quad V \in \nabla_{\mathscr{V}} \cup \{*\}$

Here, n *represents an actual node of* V_G *while* N, P, A *and* V *are labels (or label variables) of nodes, ports, attributes and values respectively.*

We say that $lv_G(n) \cong lv_H(m)$ for two given port graphs G and H if all the family functions results are equal for n and m with the added flexibility that a variable label can be equal to any label, that $*$ is equal to any value or attribute label. This definition of $\cong$ will be used in the Port Graph Morphism to follow.

- *$E_G \subseteq \{\langle (v_1, p_1), (v_2, p_2)\rangle \mid v_i \in V_G, p_i \in Interface(lv_G name(v_i))\}$ is a finite multi-set of edges where $\langle (v_1, p_1), (v_2, p_2)\rangle$ is an unordered pair (edges are undirected). E_G needs to be a multiset since two nodes can be connected by more than one edge on the same ports.*

To be able to track individual nodes, each node is given an identity (the nature of this identity is determined by the implementation, for example the PORGY tool described in Chapter 7 gives each node a unique identifier). When new nodes are added to a port graph, the implementation will make sure they are assigned identities that are not currently in use in that port graph. This will guarantee that nodes in a port graph, while being able to have the same node label, will never have the same identity.

Attributes and Values

As defined in 2.1.1, a port may have a set of pairs of attributes and values which can be used to represent a state (for instance, active/inactive or principal/auxiliary) or other properties (for example weight or colour). Similarly, nodes can also have attributes representing properties such as colour, shape, root, leaf...

Two nodes with the same node label must have the same set of attributes (in the same way that they must have the same set of port labels) however the values of those attributes can be different. Similarly, ports with the same port label (belonging to nodes with the same node label) must have the same of attributes but their values can be different.

We denote $a = v$ to mean that the attribute label a has been assigned the value label v. For example $active = True$, $colour = blue$ and $size = *$ (in the latter example, this would mean we do not care what the value of $size$ is, just that the attribute exists).

We assume that for implementations values are basic data types such as *strings*, *int*, *float*, *bool*. Values are not computable expressions so therefore we have no issues with decidability, non-termination, and non-determinism.

These attributes may be used for visualisation purposes and are essential for the definition of the `Property()` operator introduced in Section 5.2; they are later illustrated in examples.

Figure 5.2 in Section 5.2 formally defines the language for attributes and values to be used with the `Property(,)` operator.

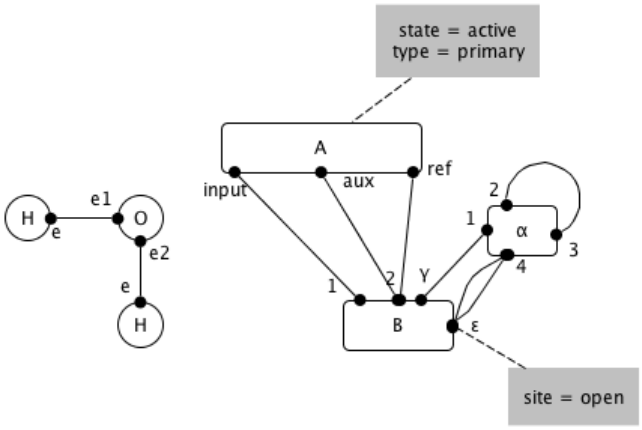


Figure 2.1: Some examples of port graphs. α denotes a variable node and γ and ϵ denote two variable ports.

Graphical Convention

The view of a port graph used throughout this thesis is illustrated in Figure 2.1. On the left we see the representation of a water molecule. The two nodes labelled H are hydrogen atoms that each have have a port labelled e (representing the electron they each have). The oxygen molecule is labelled O and has two valence electrons, represented by the ports e1 and e2. Here we represent the bonds between two atoms (their sharing of one of their electrons each) using edges.

On the right we see a more abstract port graph where we see that two different ports can be connected to the same third port, two ports of a same node are connected and that two ports can have more than one edge connecting them.

Sets of pairs of attributes and values are represented by grey boxes that are connected by a dashed line to a node or a port. Typically we will not display the attributes boxes unless they are required for a rewriting rule or to explicitly show a matching of a rule (See Figure 2.2 for an example of such a rule).

Port Graph Morphism

Let G and H be two port graphs over the same *p-signature* ∇ (Defined at the start of this section). A *port graph morphism* $f : G \to H$ maps nodes, ports, edges, node attributes and values, and port attributes and values of G to those of H such that all labels are preserved, the attachment of edges is preserved and the set of pairs of attributes and values for nodes and ports are also preserved. We say that G and H are *isomorphic* if $f : \langle f_V : V_G \to V_H, f_E : E_G \to E_H \rangle$ is bijective with:

- $f_V : V_G \to V_H$, a mapping from the node set of G to the node set of H such that if $n \in V_G$ and $f_V(n) \in V_H$ then $lv_G(n) \cong lv_H(f_V(n))$. (See Definition 1 for an explanation of $\cong$)
  This ensures that each corresponding pair of nodes between G and H have the same node label, same set of port labels (each with the same set of attribute labels and associated value labels) and the same set of attribute labels and associated value labels.

- $f_E : E_G \to E_H$, a mapping from the multi-set of edges of G to the multi-set of edges of H such that if $\langle (v_1, p_1), (v_2, p_2) \rangle \in E_G$ then $\langle (f_V(v_1), p_1), (f_V(v_2), p_2) \rangle \in E_H$.

Port Graph Rewriting

We denote a *port graph rewrite rule* $L \Rightarrow R$ as a port graph by defining a node (called *arrow* node) that encodes the correspondence between the nodes of L and the nodes of R (both subgraphs of the rule).

Definition 2. *A port graph rewrite rule $L \Rightarrow R$ is a port graph consisting of:*

- *Two port graphs L and R over the p-signature ∇, the* left-hand-side *and* right-hand-side *respectively.*
- *an* arrow *node ($\Rightarrow$) that has a set of ports.*
- *a set of* arrow-edges *that each connect a port of the* arrow *node to a port of a node in L or R. This set must satisfy the condition that all the ports of the* arrow *node must be connected to either one or two* arrow-edges.
  Each port of the arrow node connected to a single arrow-edge *must be connected to a port on the* left-hand-side. *Also, each port of the arrow node connected to two* arrow-edges *must satisfy one of the following conditions regarding its two* arrow-edges*:*

  1. *The port is connected to a port p_1 of a node on the* left-hand-side *and a port p_2 of a node on the* right-hand-side.
     *This forms a bridge between both sides so that the rewiring (which will be defined later) will take the non-*arrow-edges *of p_1 and connect them to p_2.*
  2. *The port is connected to two ports p_1 and p_2 each from a node on the* left-hand-side. *This particular rewiring takes all the ports that are connected to p_1 and creates an edge for each of those ports to each of the ports connected to p_2.*
  3. *The port* **cannot** *be connected to two nodes on the* right-hand-side.

The set of *arrow-edges* is used to control the rewiring that occurs during the rewriting. When the correspondence between ports in the left- and right-hand sides of the rule is obvious we omit the ports and edges involving the arrow node. The *arrow* node is used to avoid dangling edges during rewriting (see [46, 29]). More details are given in Figure 2.2, where the *arrow-edges* are dashed or dotted lines.

This definition generalises the original definition given in [6], by alowing case (2.) above. This will allow us to easily define Interaction Nets as a particular case of port graph rewriting rules.

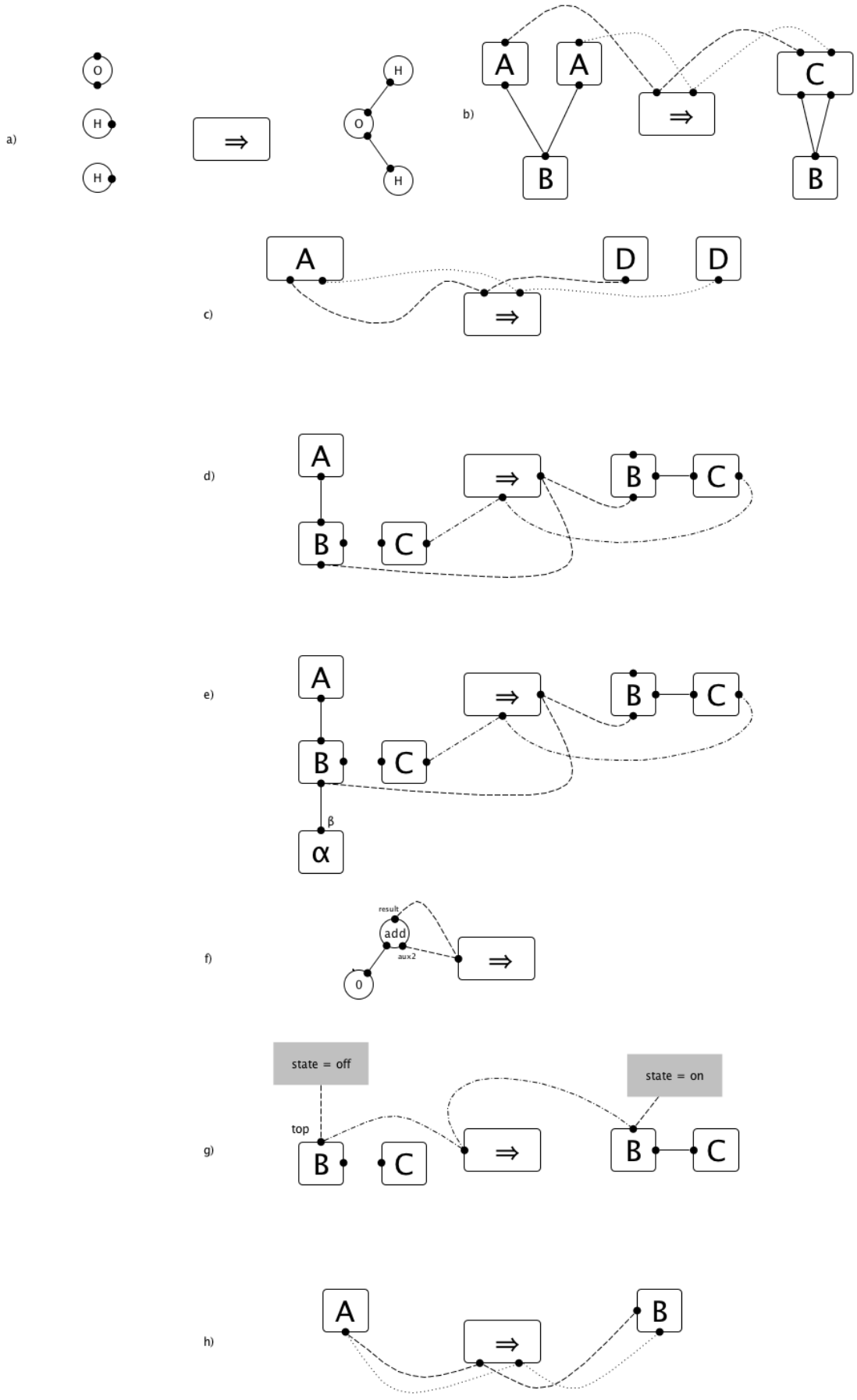


Figure 2.2: Seven examples rules, different line types are used for clarity only. α is used to denote a variable node and β denotes a variable port.

Figure 2.2.a shows a rule that takes the three unconnected atoms needed for a water molecule and creates the two necessary edges to connect both hydrogen atoms to the oxygen atom. Figure 2.2.b shows a rule that merges two nodes into one node, preserving the two edges by rewiring them to different ports. Figure 2.2.c shows a rule that splits a node A into two D nodes while preserving edges. Figure 2.2.d shows a rule that deletes a node and an edge and creates a new edge between the remaining nodes. Figure 2.2.e shows a rule similar to the one in Figure 2.2.d but here the bottom port of B must have an edge connecting it to any port on any node (expressed by using the variables α and β) as opposed to the rule in Figure 2.2.d where we do not care if the bottom port of B is connected to another port or not. Figure 2.2.e shows a rule that will rewire the graph, taking whatever is connected to the *result* port and using an edge to connect it whatever the *aux2* port is connected to. Figure 2.2.g shows a rule where a B node can be connected to a C node if the former's *top* port has its *state* attribute set to the value *off*. When the connection is made, the *top* port's *state* attribute is set to *on*. Figure 2.2.h changes an A node to a B node, and makes sure that everything that was connected to the A node's port is now connected to both ports of the B node.

Definition 3. *Let $L \Rightarrow R$ be a port graph rewrite rule and G a port graph. We say a match $g(L)$ of the left-hand-side is found if:*

- *There is an injective port graph morphism g from L to G; hence $g(L)$ is a subgraph of G.*
- *For each port in L that is connected to no edges in $L \Rightarrow R$ its corresponding port in $g(L)$* cannot *be in the set of edges of G.*

This last point will enable us to dictate the following about each port in the left-hand-side of a rule:

- The port *must* be free. This is done by not connecting any edges (arrow-edges or otherwise) to it.
- The port *must* be connected, done by using a variable node with one variable port and connecting them together.
- The port can be either free or connected. This is done by connecting the port to a port on the arrow-node. If this arrow node is also connected to another port then a rewiring will happen (see below), if not then no rewiring will be done and the edge connected to this port will be destroyed.

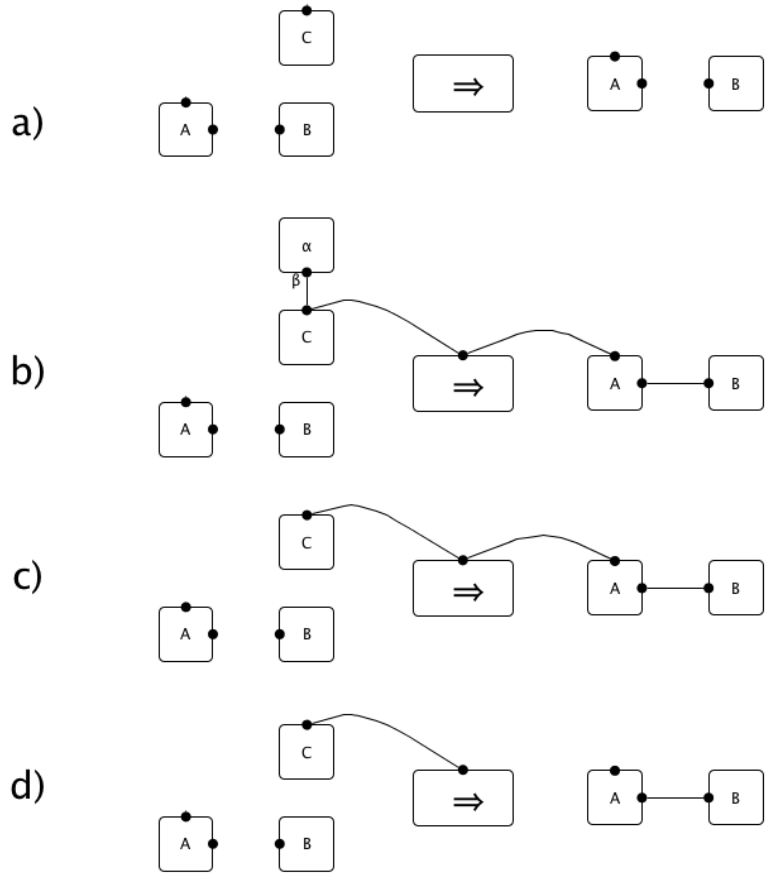


Figure 2.3: Four examples of rules. α and β are variable labels.

Figure 2.3.a shows a rule where all the ports of the left-hand-side must be free. Figure 2.3.b shows the same rule but here the port of the C node must be connected to something (expressed by using a variable node and port) and in this case the something it is connected to will be rewired to the top port of A. Figure 2.3.c is again the same rule but where we do not care whether the port of the C node is connected or free but if it is connected then rewire that edge to the port on the A node of the right-hand-side. Figure 2.3.d is the same as the previous rule but in this case, no rewiring is done if the port of the C node is connected.

Definition 4. *A* rewriting step *on G using $L \Rightarrow R$, written $G \rightarrow_{L \Rightarrow R} G'$, that transforms G into a new graph G' obtained from G happens in four phases:*

- *The* build *phase where a redex $g(L)$ is found in G and an instance of the port graph rule $L \Rightarrow R$ is put into G. This creates the graph G_1*
- *The* matching *phase where we replace the sub graph L from the added rule by the redex g(L), preserving all the* arrow-edges *that connect to the* arrow *node. This creates the graph G_2.*
- *The* rewiring *phase where we take each port on the* arrow *node connected to two arrow-edges and if:*

  - *it is connected to a port p_1 of a node on the* left-hand-side *and a port p_2 of a node of the* right-hand-side *we then take all the ports connected to p_1 and connect them with an edge to p_2.*
  - *it is connected to two ports p_1 and p_2 both on nodes on the* left-hand-side *we then take all the ports that are connected to p_1 and connect each of them to each port connected by an edge to p_2.*

  Af the end of this phase, all edges connected to the arrow node are deleted.

- *The* deletion *phase simply deletes the* arrow *node and all the nodes from g(L)* **that weren't matched to a node with a variable label** *also removing any edges that were connected to their ports, thus eliminating any* dangling edges. *This creates the graph final graph G'.*

See Figure 2.4 and Figure 2.5 for a phase by phase application of the rewrite rule found in Figure 2.2.d 2.2.e respectively on a port graph. Figure 2.6 shows a phase by phase application of a rule with a variable label on one of its nodes: here the rule is forcing the top port of A to be connected (and to rewire that connection to B in the rewrite step), the right port of A to be free, and the bottom port of A to be either free or connected (but if it is connected, the edge will be destroyed during the rewriting).

The nodes of the original port graph G will be drawn with dashed lines as a visual aid.

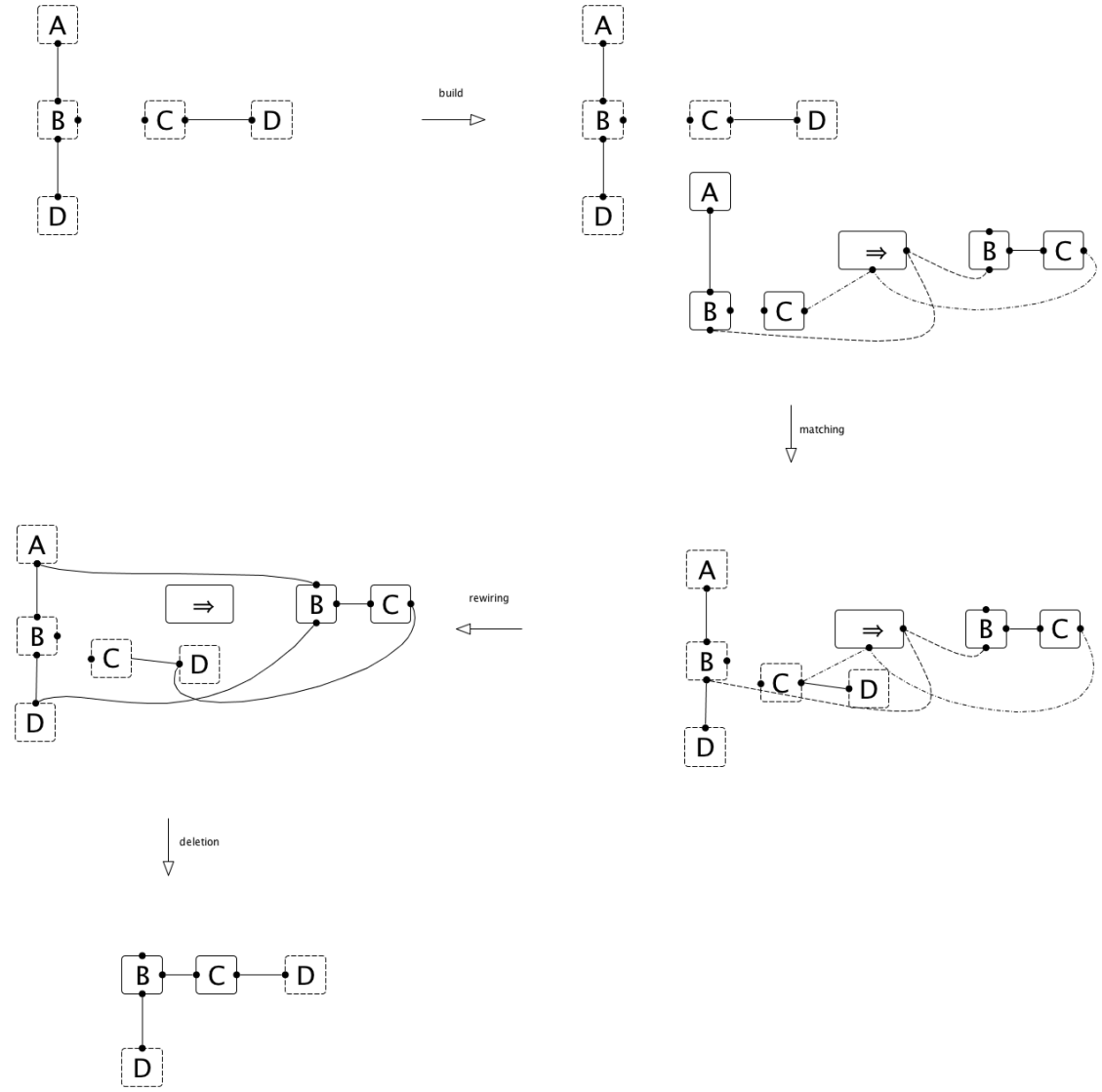


Figure 2.4: A rewrite step showing all the phases of the rewriting using the rule in Figure 2.2.d on a port graph.

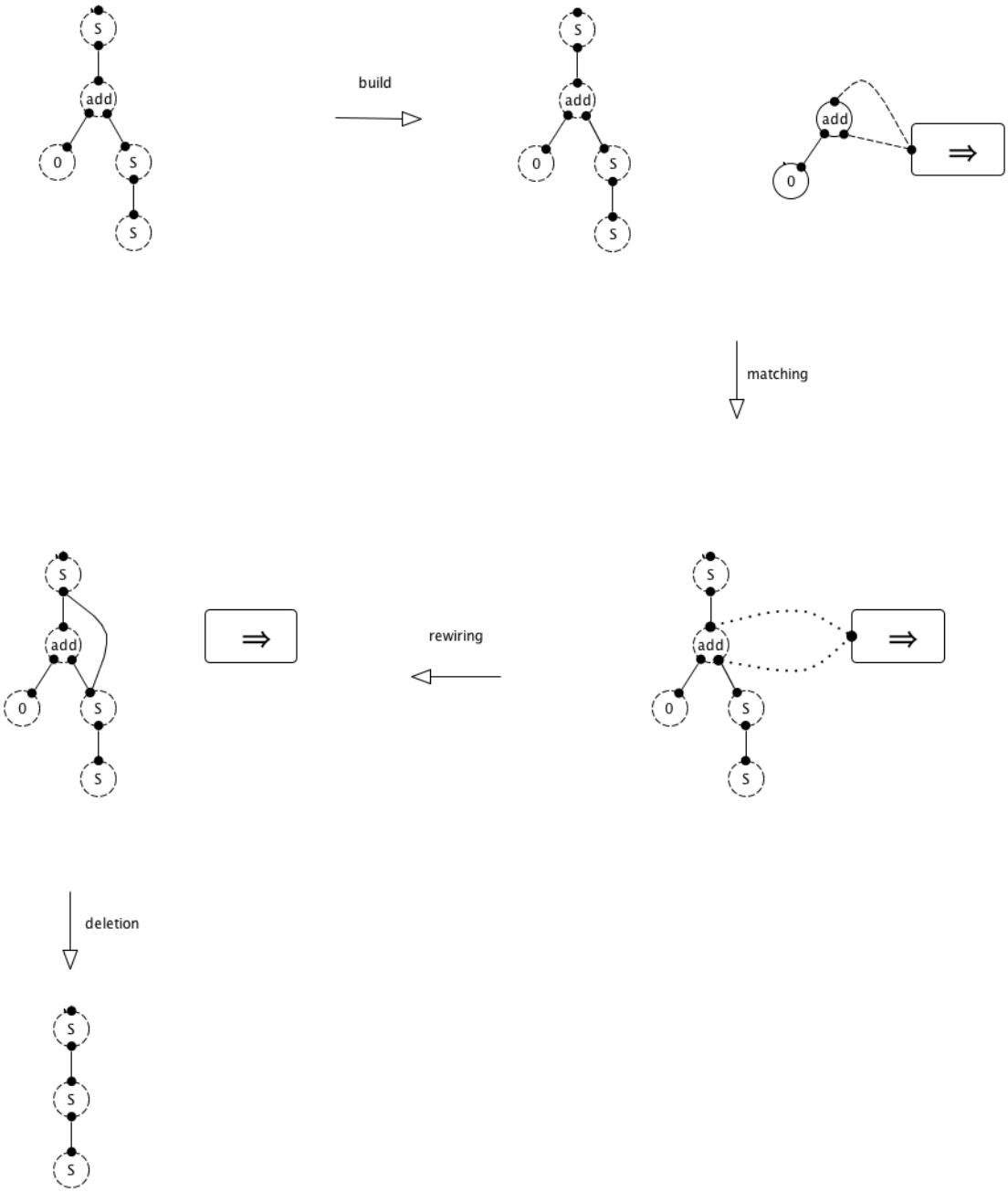


Figure 2.5: A rewrite step showing all the phases of the rewriting using the rule in Figure 2.2.e on a port graph.

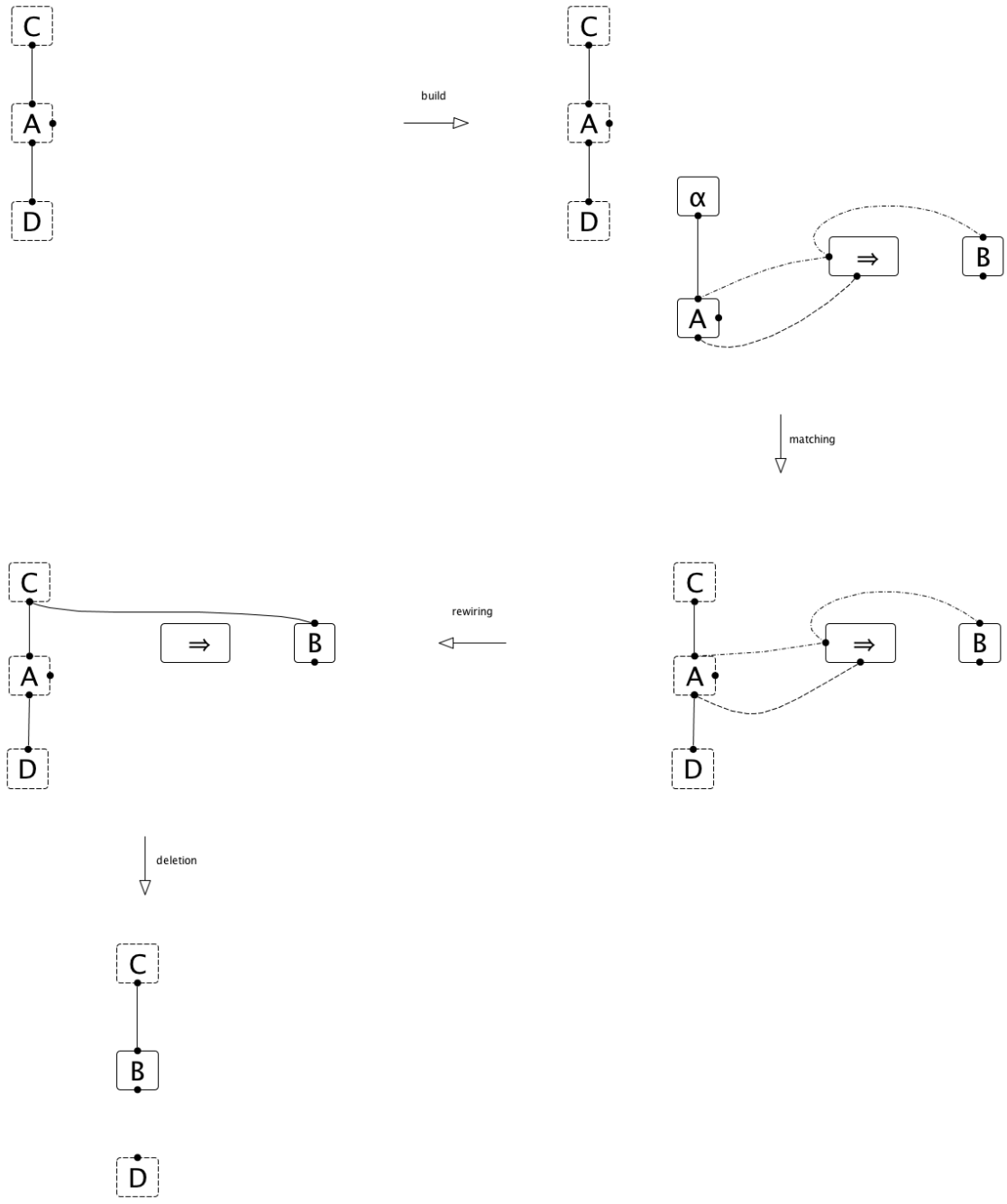


Figure 2.6: A rewrite step showing all the phases of the rewriting using a rule with a variable node label.

Several injective morphisms g from L to G may exist (leading to different rewriting steps); they are computed as solutions of a *matching* problem from L to (a subgraph of) G. If there is no such injective morphism, we say that G is *irreducible* by $L \Rightarrow R$. Given a finite set $\mathcal{R}$ of rules, a port graph G *rewrites* to G', denoted by $G \rightarrow_{\mathcal{R}} G'$, if there is a rule r in $\mathcal{R}$ such that $G \rightarrow_r G'$. This induces a transitive relation on port graphs, denoted by $\rightarrow^*_{\mathcal{R}}$. Each *rule application* is a rewriting step and a *derivation*, or computation, is a sequence of rewriting steps.

A port graph on which no rule is applicable is in *normal form*. Rewriting is intrinsically non-deterministic since it may be possible to rewrite several subgraphs of a port graph with different rules or use the same one at different places, possibly getting different results.

Summary

Port graphs present a powerful, versatile and intuitive modelling system for graphs and graph rewriting. The notion of ports is easy to understand and can be applied to many fields intuitively such as Chemistry (as protein receptors) and network modelling (as network ports) to name a few. Since nodes and ports can have attributes, it is easy to store data within port graphs and used with rewriting rules and the right kind of strategy the data can be used to guide the rewriting (You can see examples of this using the `Property(,)` operator in Sections 6.6 and 6.7). The $L \Rightarrow R$ notation for graph transformations is again intuitive and can be easily understood and used in many fields; for instance chemical reactions are written with the same $L \Rightarrow R$ notation. The inherent non-determinism is also useful since a lot of modelling needs to take non-determinism into account: in Section 6.6 pacman can sometimes move in more than one direction and so one should be picked randomly. Further details and examples of port graphs can be found in Chapters 5 and 6 and in [10, 11].

2.1.2 Interaction Nets

Interaction nets were introduced by Lafont in 1990 [52] and later used as a target language for implementation of efficient λ-calculus evaluators (see for instance [45, 56]). We give below a brief description of this formalism. Interaction Nets have their own graphical convention used throughout this section. A discussion on how Interaction Nets are a particular case of Port Graphs is given at the end of the section, complete with examples of Interaction Net rules drawn as Port Graph rules.

A system of interaction nets is specified by a set Σ of symbols with fixed arities, and a set $\mathcal{R}$ of interaction rules. An occurrence of a symbol $\alpha \in \Sigma$ is called an *agent*. If the arity of α is n, then the agent has $n+1$ *ports*: a *principal port* depicted by an arrow, and n *auxiliary ports*. Figure 2.7(a) shows a typical drawing of such an agent. Intuitively, a net N is a port graph (not necessarily connected) where the nodes are agents. The ports that are not connected to another agent are said to be *free*. The *interface* of a net is its set of free ports.

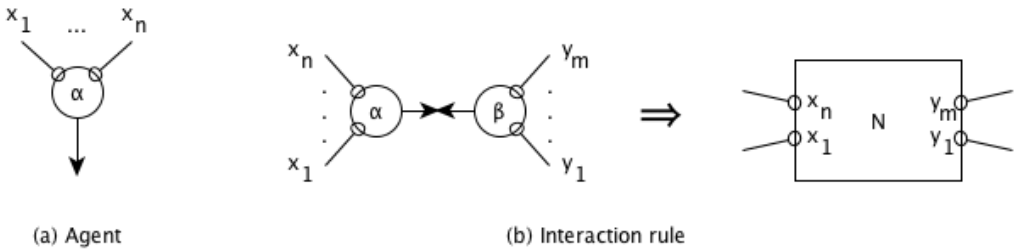


Figure 2.7: General format of an agent and an interaction rule.

Interaction Rules

Interaction rules are port graph rewrite rules where the left-hand side consists of two agents $(\alpha, \beta) \in \Sigma \times \Sigma$ connected together on their principal ports (an *active pair* or *redex*) and the right-hand side

is a net N. The diagram depicted in Fig. 2.7(b) shows the format of interaction rules (N can be any net built from Σ).

Interaction rules must have the following two conditions:

- The left and right hand sides of a rule must have the same interface, that is to say that the number of free ports on both sides must be equal.
- At most one rule can be given for each pair of agents (α, β).

The net N in an interaction rule (α, β) can contain occurrences of α or β and can also just be a *wiring* (a term used to describe an edge and is mainly used for cases where the right hand side of a rule is a subgraph containing only edges and no nodes. See Figure 2.9) as long as the number of free ports in the left hand side is even. It is possible for the active pair to have no free ports, in that case N may be (but not necessarily) empty.

A reduction using a rule $(\alpha, \beta) \Longrightarrow N$ takes an occurrence of the active pair (α, β) and replaces it by the net N. We can write $W \Longrightarrow W'$ if we can find an active pair (α, β) in W and an interaction rule $(\alpha, \beta) \Longrightarrow N$ such that W' is obtained by replacing the occurrence of the active in W with N. We use $\Longrightarrow$ for a single interaction step and $\Longrightarrow^*$ for the transitive reflexive closure of the relation $\Longrightarrow$.

Strong Confluence & Turing Completeness

Interaction nets are *strongly confluent*, that is to say that if in a net N two different reduction $N \Longrightarrow N_1$ and $N \Longrightarrow N_2$ are possible then there exists a net M such that both N_1 and N_2 can reduce to M in one step (meaning that both $N_1 \Longrightarrow M$ and $N_2 \Longrightarrow M$ are possible). Strong confluence is due to the fact that during a reduction step, the interface is preserved. This means that the reduction is local since the reduction doesn't affect the rest of the net and each agent can only be in one active pair at a time since it only has one principal port.

Interaction nets are also Turing complete as shown in Section 7.3 of [37].

Parallelism

The fact that reduction is local (see above) provides interaction nets with a very useful property: parallelism. Since reduction doesn't affect the rest of the graph (and the interface is always preserved) if there is more than one active pair, it is possible to do their rewrite steps simultaneously. Unlike most other computational models, the design of the code (or rewriting rules) do not need to be adapted for parallelism, they work naturally since parallelism is an inherent property of interaction nets.

Full Normal Form & Interface Normal Form

We say that an interaction net is in *full normal form* (also simply called *normal form*) if it has no active pairs. While *strong confluence* dictates that there is only one normal form (assuming the interaction net is terminating), there are clear multiple ways of getting to the normal form.

In some contexts, for instance when using interaction rules to program functions, we need a weaker notion of normal form corresponding to the notion of weak head normal form in λ-calculus. Interface normal form, defined in [38], can be computed by a strategy which applies rules at the interface (if possible) in order to minimise the length of the reduction sequence. Taken from [37]: "Intuitively, an interaction net is in interface normal form when there are agents with principal ports on all of the observable interface, or, if there are ports in the interface that are not principal, then they will never become principal by reduction (since they are in an open path or a cycle)."

Arithmetic Example

Natural numbers can be represented using 0 and the successor function. For instance 2 can be written as $S(S(0))$. For the addition operator, we can specify the following:

- $add(0, y) = y$
- $add(S(x), y) = S(add(x, y))$

We can code the following using three types of agents (figure 2.8) 0, S and add.

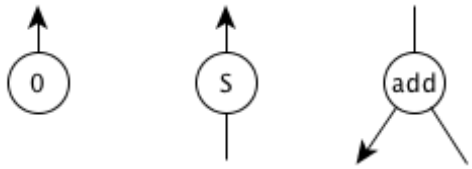


Figure 2.8: Three agent types needed to code addition.

We then create two interaction rules, one for $add(0, y) = y$ (figure 2.9) and one for $add(S(x), y) = S(add(x, y))$ (figure 2.10).

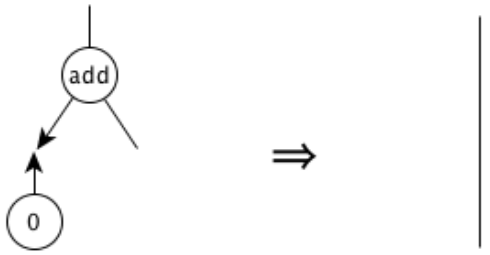


Figure 2.9: First *add* rule.

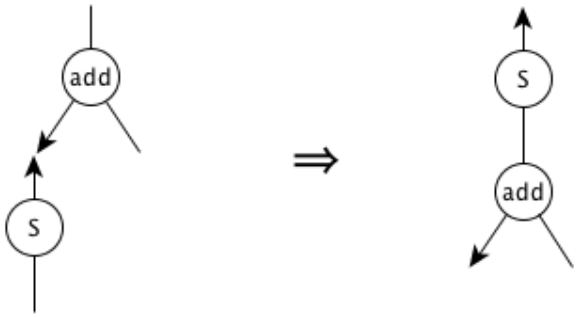


Figure 2.10: Second *add* rule.

Consider the starting net in figure 2.11 representing $2 + 1$; Figure 2.12 shows the application using the second rule twice and then one application of the first rule. The result we get is $S(S(S(0)))$ which is 3.

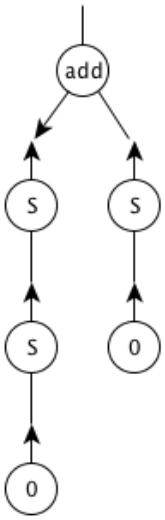


Figure 2.11: Starting net expressing $2 + 1$.

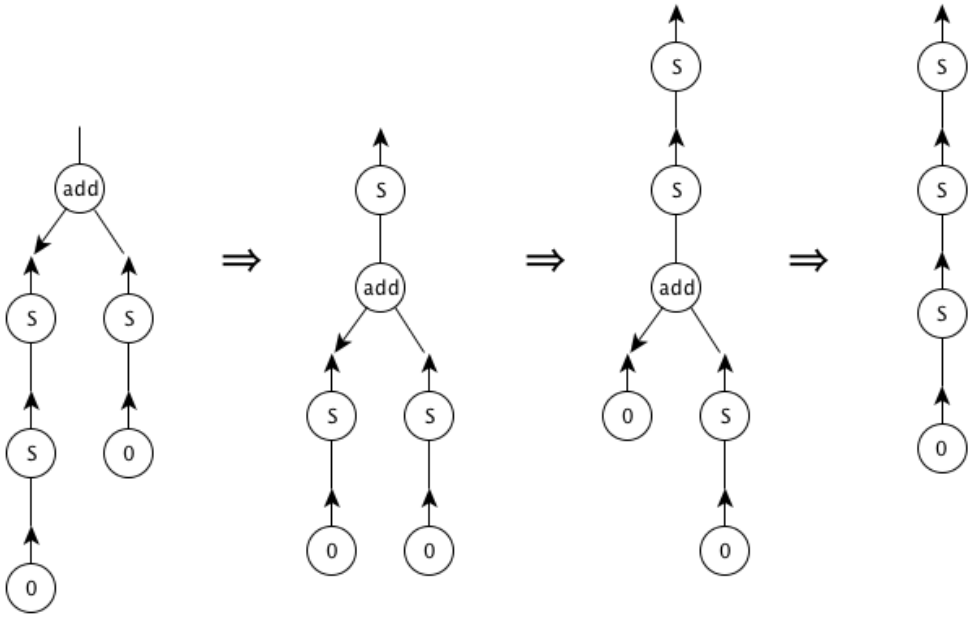


Figure 2.12: Three successive rewrites.

We can further extend this example to incorporate *multiplication* by adding three more agents (figure 2.13): the ϵ, δ and *mult* agents:

- ϵ is called the *erasing agent* and erases the agent it is interacting with and then propagates to the erased agent's auxiliary ports (figure 2.14).
- δ is called the *duplicator agent* and copies any agent it is interacting with then propagates to the copied agent's auxiliary ports. (figure 2.15).
- *mult*, like *add* is the agent used to represent multiplication.

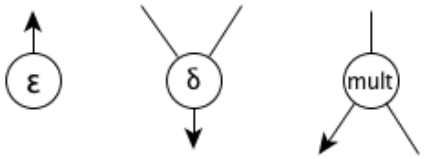


Figure 2.13: Three agent types needed to code multiplication.

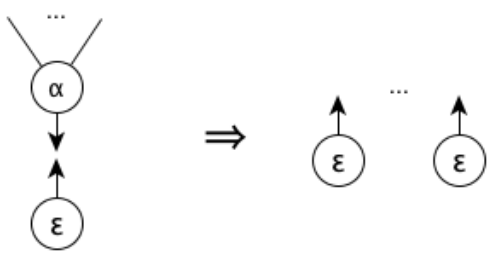


Figure 2.14: The ϵ rule.

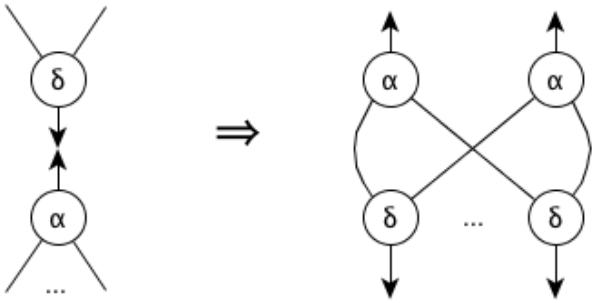


Figure 2.15: The δ rule.

To implement multiplications we can represent the following:

- $mult(0, y) = 0$ (figure 2.16)
- $mult(S(x), y) = add(mult(x, y), y)$ (figure 2.17)

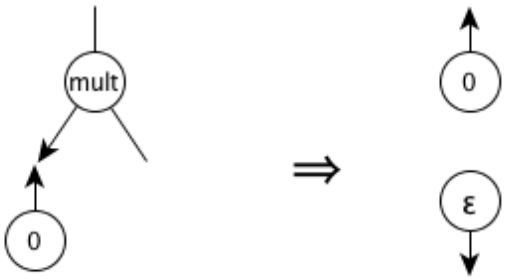


Figure 2.16: The first *mult* rule.

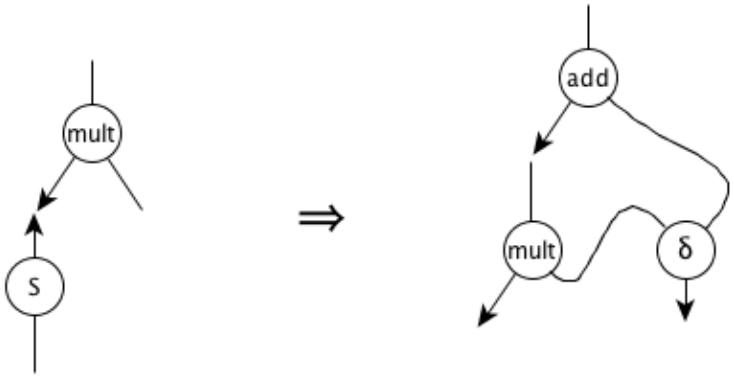


Figure 2.17: The second *mult* rule.

Interaction Nets as a Particular Case of Port Graphs

Interaction nets can be implemented with port graphs since interaction net agents have a set of distinct ports, something that can be represented in a port node. The principal port of an agent can simply be a particular port of a port node that is labelled *principal*.

Most interaction net rules can be written as port graph rules such as the rule in Figure 2.18. Rules that use wiring (like in Figure 2.9) can also be written with port graph rewriting rules (with the extension definition given earlier). The PORGY tool allows the creation of right hand sides that just contain wirings (visually, this is represented by using a specific colour for those edges when drawing the rule). For example the rule in Figure 2.9 is represented as a port graph rule in Figure 2.19.

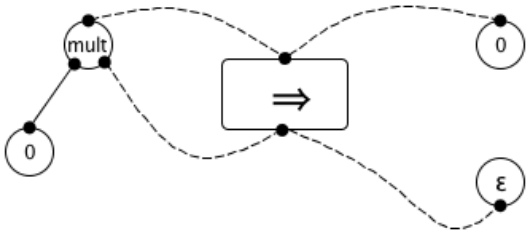


Figure 2.18: The rule in Figure 2.16 redrawn as a port graph rule.

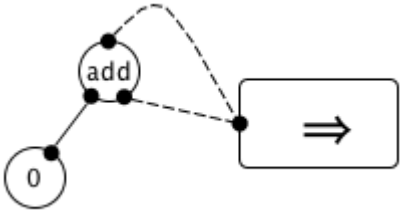


Figure 2.19: The rule in Figure 2.9 redrawn as a port graph rule.

Summary

Interaction nets provide an interesting way to model rewriting with a strong restriction on rules that provides strong confluence and parallelism.

Interaction rules can be seen as a particular kind of port graph rewriting rules with constraints that ensure good computational properties: pattern-matching is particularly easy in interaction nets, since patterns are graphs containing just two nodes, and the transformations are local and strongly confluent.

Chapter 3

Related Work

3.1 Graph and Hypergraph Rewriting

There exists different formalisms to describe and represent graphs and graph transformations. Each formalism has a different set of properties that makes it better suited to certain applications. Below is a description and review of some of the main formalisms that exist.

3.1.1 Hypergraphs and Hyperedge Replacement

Hypergraphs

Hypergraphs[34] are graphs where an edge can connect any number of vertices. Let C be an alphabet such that every symbol $A \in C$ has a type $type(A) \in \mathbb{N}$, hypergraphs are defined as a tuple (V, E, att, lab, ext) where (taken from [5]):

- V is the set of vertices of the hypergraph
- E is a finite set of *hyperedges*.
- att: $E \to V^*$ is a mapping assigning a sequence of attachment nodes att(e) to each $e \in E$
- lab: $E \to C$ is a mapping that labels each hyper edge such that $type(lab(e)) = |att(e)|$
- $ext \in V$ is a sequence of pairwise distinct external nodes.

For example, we define the V and E of a hypergraph H drawn in Figure 3.1 as:
$V = \{v_1, v_2, v_3, v_4, v_5, v_6\}$ and $E = \{e_1, e_2, e_3\}$
where $e_1 = \{v_1, v_2, v_3\}, e_2 = \{v_1, v_3, v_5\}, e_3 = \{v_5, v_6\}$

Hyperedge Replacement

In [5], hyper edge replacement is described as:

Let H, H' be hyper graphs, $e \in E_H$ with $type_H(e) = type(H')$. Then the replacement of e by H' in H yields the hyper graph $H[e/H']$, which is obtained as follows:

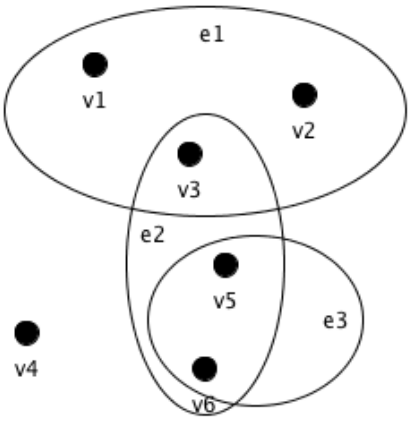


Figure 3.1: An example hypergraph.

- Build $H \setminus e$ by removing e from H.
- Take the disjoint union of $H \setminus e$ and H', in other words rename nodes and edges appropriately; the external nodes are those of H.
- For all $i \in 1, ..., type_H(e)$ identify the i-th external node of H' with the i-th attached node of e.

Also, [34] states that:

Hyperedge replacement enjoys some nice properties well-known from other rule-based formalisms. First of all, we have a sequentialization and parallelization property. It does not matter whether we replace some hyperedges of a hypergraph one after another, or simultaneously. The second property is confluence. Hyperedges of a hypergraph can be replaced in any order, without affecting the result. (In fact, this follows already from the sequentialization and parallelization property. If we replace hyperedges simultaneously there is no order among them at all.) The last and maybe most important property is associativity. If a hyperedge is replaced and afterwards a hyperedge of the new part is replaced with a third hypergraph, the same is obtained by first replacing the latter hyperedge and then replacing the first one with the result.

While hyper graphs and hyperedge replacement can be used to model a diverse range of systems, it is not as intuitive as using a graph system with ports. Ports can be simulated by using hyperedges as the nodes and vertices to connect these nodes together but make it unnecessarily complicated and unintuitive compared to port graphs. The confluence and inherent parallelism found in hyper edge replacement (much like interaction nets) is a very useful property to have since strategy language defined in Section 5 supports application of rules but again, if such properties are needed, a specific case of port graphs can be used (for example, it is possible to simulate interaction nets using port graphs).

3.1.2 Graph Grammars

Graph Grammars were designed as an extension of formal grammars on strings to grammars on graphs.

Inspired by [35], a graph grammar is a grammar rewriting system expressed as a quadruple: $GG = (N, T, P, S)$ where given an arbitrary but fixed set of labels C:

- $N \subseteq C$ is a set of *nonterminals*.
- $T \subseteq C$ is a set of *terminals* with $T \cap N = \emptyset$.
- P is a finite set of *productions* over N.
- $S \in N$ is the start graph.

The *language* $L(GG)$ generated by GG is $L_S(GG)$ (where for all $A \in N$, $L_A(GG)$) is a set of all terminally labelled graphs that are derivable from S by applying productions.

Graph grammars can take the role of generation where they take the starting graph and use the productions to generate a final graph. They can also take the role of recognition where a final graph is given and the graph grammar sees if that graph can be created from the starting graph using the given rewrite rules.

With Graph Grammars, the focus is more to generate a *language* or to verify if a particular graph is a valid for a certain grammar. Also, there is no control over the *productions* in P and how and when they are applied.

3.1.3 Interaction Nets

See Section 2.1.2 for a definitions and explanation of interaction nets.

Interaction nets hold some interesting properties like strong confluence and parallelism but are more *low-level* than port graphs and so they are not as intuitive to use, that is to say that the strong restriction of the *principal port* does give interaction nets some interesting features but makes creating some rules more complex because of this restriction. Also, since interaction nets are a particular case of port graphs (a restriction on rewriting rules), we can easily model interaction nets using port graphs if needed.

3.1.4 Summary

Port graphs give us a more natural and easy to understand representation of graphs and graph rewriting over other formalisms. While other formalisms like hypergraphs and hyperedge replacement have useful properties such as parallelism, those properties can be simulated with port graphs.

3.2 Graph rewriting Tools

Several tools are available to create and edit graphs, and some of them allow users to model graph transformations. Below we review some of the systems that share some common goals with PORGY (described in Chapter 7) .

3.2.1 GROOVE

GROOVE [68] is centred around the use of simple graphs for modelling the design-time, compile-time, and run-time structure of object-oriented systems. The model transformations and the operational semantics is defined on graph transformations. Model checking is used to verify model transformations and dynamic semantics through an (automatic) analysis of the resulting graph transformation systems. GROOVE's rewriting rules are not defined as $L \Rightarrow R$ but as one graph where the nodes and edges can be one of four types:

- *readers* : these need to exist to apply the rule and survive the rewriting.
- *erasers* : these must also exist to apply the rule but are removed from the graph during the rewriting.
- *creators* : that are created during the rewriting
- *embargoes* : that mustn't exist for the rewriting to happen.

While this is equivalent to a $L \Rightarrow R$ definition of rules, it appears more complicated to understand since it has essentially merged the L and R graphs into one same graph and uses colour coding to show what is needed,banned, added and removed.

The GROOVE tool set includes an editor for creating graph production rules, a simulator for visually computing the graph transformations induced by a set of graph production rules, a generator for automatically exploring state spaces, a model checker for analysing the resulting graph transformation systems and an imaging tool for converting graphs to images. Visualisation is not the main objective, and after each rewrite step the user must update the layout of the graph by hand. GROOVE can simulate 20,000 states in 30 minutes and is therefore useable for large systems (much like PORGY) but doesn't allow us to do complex renderings in 3D which would allow for some interesting and useful visualisations.

GROOVE permits to control the application of rules, via a control language with sequence, loop, random choice, try()else() and simple (non recursive) function calls. These are similar to PORGY's constructs, but the main difference is that GROOVE's language does not include the notion of position, it is not possible to specify a focus (and evolve and move this focus) for the application of rules within the language. Tracing - an important feature of PORGY - is possible with GROOVE through *state space exploration*. A unique feature of GROOVE is the *embargoes* type found in the rewrite rules: it is possible to explicitly states within a rule that a particular subgraph *mustn't* exist for the rule to be applied. Port graphs do not have this feature written within its rule formalism but the strategy language described in Chapter 5 (with the use of the `Property(,)` operator and the `if()then()else()` construct) allows us to do the same.

3.2.2 Fujaba

The Fujaba [61] Tool Suite is an Open Source CASE tool providing developers with support for model-based software engineering and re-engineering. The Fujaba project aims at developing and extending the Fujaba Tool Suite and thus offering an extensible platform for software engineering researchers. It combines UML class diagrams [41, 23], UML activity diagrams, and a graph transformation language (so called Story Patterns) and offers a formal, visual specification language

that can be used to completely specify the structure and behaviour of a software system under development. Graphs and rules are used to generate Java code. Fujaba is heavily focused UML based models and therefore doesn't offer the versatility to model diverse systems, which is an important aspect of the PORGY tool and the strategy language described in Section 5. Fujaba has a basic strategy language, including conditionals, sequence and method calls. There is no parallelism, and again one of the main differences with the strategy language described in Section 5 is that Fujaba does not include a notion of location to guide the rule application.

Fujaba's interface is quite visual allowing users to draw UML diagrams intuitively within its GUI, an important feature for PORGY and graphPaper that will decrease the learning curve for less technically minded users.

3.2.3 AGG

AGG [36] is a rule based visual language supporting an algebraic approach to graph transformation. It aims at the specification and prototypical implementation of applications with complex graph-structured data. AGG may be used as a general purpose graph transformation engine in high-level JAVA applications employing graph transformation methods. The application of rules can be controlled by defining *layers* and then iterating through and across layers, simulating priority application. No further control seems possible.

Rules can also have *negative application conditions* where a rule can only be applied if a certain *structure* (subgraph) doesn't exist, something that can be simulated with the strategy language defined in Chapter 5 using the `Property(,)` operator and the `if()then()else()` construct.

The tool itself allows users to create graphs and rules and code Java expressions. It also represents the interpretation and validation visually.

Again, there is no notion of position and there is no control on the search for redexes.

3.2.4 PROGRES

The PROGRES [69] project works on the theoretical foundations as well as the practical implementation of an executable specification language based on graph rewriting systems (graph grammars). The aim is to combine EER[60]-like and UML-like class diagrams for the definition of complex object structures with graph rewrite rules for the definition of operations on these structures.

The PROGRES editor is both a graphical and a textual editor since it mixes more graphs and text in its language. It also has a type checker for the language's static semantics.

PROGRES allows users to define the way rules are applied (it includes non-deterministic constructs, sequence, conditional and looping) but it does not allow users to specify the position where the rule is applied. It is a very expressive language and includes a tracing functionality through backtracking, something very useful for debugging and static analysis.

3.2.5 GrGen

GrGen.NET [44] is a programming tool for graph transformation designed to ease the transformation of complex graph structured data as required in model transformation, computer linguistics,

or modern compiler construction, for example. It is comparable to other programming tools like parser generators which ease the task of formal language recognition or databases which ease the task of persistent data storage and querying.

The tool works mainly from textual files and then transforms that text into graphs. Our motivations are different: we are looking for a more intuitive method where users can create the graphs directly into a visual environment instead of learning a textual language to represent graphs.

GrGen's rule application language includes sequential, logical and iterative control which makes it Turing complete. Users can also use variables for sub graphs involved and resulting from rewrites. Some low level positioning and focusing (defined in Section 5.1.2) can be achieved with the use of a *visited* property. These properties can be applied to nodes and edges and can be used in rules. Our strategy language and rule system doesn't use properties in the rules themselves, but properties can be used to filter the main graph into subgraphs that can then be used for focusing.

The tool performs well in terms of the benchmark shown in [44] which is essential for pattern matching in large graphs. Fast pattern matching is a very important feature and is a large priority for the PORGY tool.

3.2.6 GReAT

GReAT (Graph Rewriting and Transformation) [16] is a tool for building model transformation tools. First, one has to specify the metamodels of the input and target models using UML style class diagrams and give rules to specify the transformation. Using only UML style class diagrams makes it difficult to use the tool for different kinds of modelling such as chemical reactions and functional programming.

Rules are pairs of typed, attributed graphs. The pattern-matching algorithm always starts from specific nodes called "pivot nodes". Rule execution is sequenced and there are conditional and looping structures allowing some basic control.

3.2.7 Summary

While all these tools have interesting features, they are all very specific to a certain application such as UML diagrams and object-orientated systems. Our aim with PORGY is to create a tool that can handle a varied range of models, anything from arithmetic to constructing fractals.

These tools also have a sometimes limited control mechanism and none seem to really take positioning in a graph into account. The strategy language designed to go with PORGY has been built with powerful yet intuitive and well established strategic constructs and has a focus on positioning within the graphs and the possibility of designing traversals for non-tree graphs.

3.3 Strategic Term & Graph Rewriting Tools

3.3.1 ELAN

ELAN[24] is a rule-based programming language controlled by strategies that rewrites on terms.

ELAN has control methods such as sequential composition, iteration, deterministic and non-deterministic choice operators to help guide the rewriting.

The main originality of ELAN is the ability to see if a function returns only one, at least one or many results.

In ELAN, a rule is the most elementary strategy (called a primal strategy) and operators exists (some explained below) that take strategies (primal or not) as arguments to form an *bigger* strategy.

Some variable arity operators like $dk(S_1, ..., S_n)$ (don't know choose) and $dc(S_1, ..., S_n)$ (don't care choose) return a set of results for each strategy S and picks one non-failing strategy and returns all its results respectively. Such operators are useful as *All* and *One* strategic concepts and can be used for traversals of terms.

The $first(S_1, ..., S_n)$ operator selects the first strategy that won't fail and returns all its results, allowing the user to prioritise certain strategies over others.

ELAN also has a backtracking ability to help with the non deterministic nature of rule-based programming.

3.3.2 TOM

TOM[17] is a language that extends JAVA, providing it with high level constructs to model rewriting concepts. The motivation behind TOM was to allow large scale applications to use rewriting techniques.

TOM was created after years of developing ELAN with a priority on integration, in this case with JAVA. The `%match` construct is added into JAVA to enable users to use rewriting. `%match` takes a list of expressions and contains a list of rules. Rules are defined by:

- *a left hand side* built upon constructors and fresh variables.
- *a right hand hand side* that is *not* a term, but a JAVA statement that is executed when the pattern matches the subject. The backquote construct (‘) allows terms to be built and returned if needed.

An example from [17] showing the code needed to represent addition:

```
public Nat plus(Nat t1, Nat t2) {
    %match(t1,t2) {
        x,zero() -> { return ‘x; }
        x,suc(y) -> { return ‘suc(plus(x,y)); }
    }
}
```

To control the rewriting, TOM is inspired by ELAN[24] and Stratego[75][26] and uses a high-level strategy language defined by combining low-level strategic primitives such as `Sequence(s1,s2)`, `Choice(s1,s2)`, `All(s)` and `One(s)`. Using these primitives, constructs such as `Try(s)` and `Repeat(s)` are possible, giving TOM a powerful and versatile control method.

With TOM, the notion of position is more trivial that the notion defined in Chapter 5 since we are always dealing with terms and traversals such as `TopDown` can be easily created.

Because TOM is added into JAVA and deals with terms, it is entirely textual. This means that while it is possible to model visual things such as chemical reactions, it isn't very intuitive to define the rules and there is no built in graphical output to visualise and analyse the results.

3.3.3 Stratego

Stratego[75, 26] is a language and toolset for program transformation.

Similarly to TOM, primitive strategic constructs (such as sequence, negation, choice and recursive closure) allow users to create more complex strategies. Added to those are *term traversal primitives* that deal with the notion of position within terms.

Since Stratego deals with terms, all rules and strategies are purely textual and - much like TOM- the modelling of more visual systems is possible but not intuitive or efficient.

3.3.4 GP

GP [67] is a rule-based, non-deterministic programming language. Programs are defined by sets of graph rewriting rules and a textual expression that describes the way in which rules should be applied to a given graph. Rules are given names (and both nodes and edges can have labels, and labels could be constants or variables, ie we can define rule "schemes" with conditions on the instantiation of labels).

The simplest expression is a set of rules, and this means that any of the rules can be applied to rewrite the graph. The language has three main control constructs: sequence, repetition and conditional (if-then-else), and it has been shown to be Turing complete. GP has no built in notion of position but can use variable labels in rules as a low level alternative. (See Sierpinski's Triangle Generation example in [57] and [71]).

GP has a JAVA based graphical editing environment that allows the creation, editing and execution of graphs, rules and strategies. The editor is purely a front end to facilitate creation and design since it then converts whatever the user has created into GP's textual language.

GP uses the York Abstract Machine[58] (YAM) to execute its programs. YAM is a backtracking graph-transformation machine which means that if at some point no rule can be applied, it backtracks to the nearest point where there was a choice of redex (users cannot easily handle the derivation tree or change the backtracking algorithm). YAM allows for reasonably fast execution, only slowing down if there is a large non-deterministic search space (something that can be avoided by programming carefully).

3.3.5 Summary

The described strategic tools each provide a strong strategy language to control rewriting. From the term rewriting strategies, we can take a lot of inspiration for control operators but not all their notions can be directly applied to graphs (especially when dealing with non-tree graphs); for example traversals.

GP provides a very powerful control mechanism and can use labels to create traversals which, while functional, is not as direct of an approach as the one seen in Chapter 5 and could possibly have a steeper learning curve for users with less programming experience. For example, compare the Sierpinski's Triangle Generation using GP in [67] with the one described in Section 6.5.

Chapter 4

graphPaper : A Tool to Create Graph Rewriting Systems

4.1 Description

graphPaper[39] is a cross-platform tool designed to create and edit port graph rewriting systems and its rules in a visual and intuitive way. Its purpose is to provide a focused environment to create graphs and rules before transferring them to PORGY to execute rewriting and strategies. The following sections will explain the design philosophy behind graphPaper and how the user interacts with the tool. The implementation is then described as well as its connection to the PORGY tool described in Chapter 7.

4.2 Design Philosophy

In the past few years, new technologies have shown that new Human Interface Devices (HID) can be an efficient way of dealing with digital information. Apple's touch screens used in their phones and tablets give a very direct and physical method to interact with data on a screen and Nintendo has shown with the Wii that controls that mimic real life motions are very accessible. These methods of interaction are very intuitive and have been hugely popular across all sorts of demographics since it lets users interact with on screen data in a similar way as they would with the same data in a non-digital form, for instance pushing a document on a touchscreen to pan the view or swinging a remote to make a character swing a sword.

This is the driving force behind the graphPaper tool: the creation process of a graph and its rules should be as similar as possible to how we could create them on paper but with the added dynamic benefits of computers.

4.2.1 Interaction

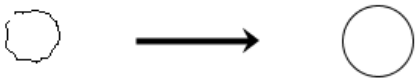
The following section will describe how the user will interact with graphPaper to create, edit and delete nodes, ports and edges, and to create rules.

For port graphs, nodes, ports and edges are the three types of data that the user will interact with. This allows the tool to have a simple interface. There is no need to have a toolbar or icons to select which type of data we want to interact with. It is possible for the tool to deduce what the user is interacting with based on the context of the interaction. For example, edges need to be created between ports so cannot be created independently. This means that if the user is trying to work on an empty part of the view, they are trying to interact with nodes (in this case, create one). When the user tries to interact with ports of a node, then we know that the goal is to create or edit edges to and from that node.

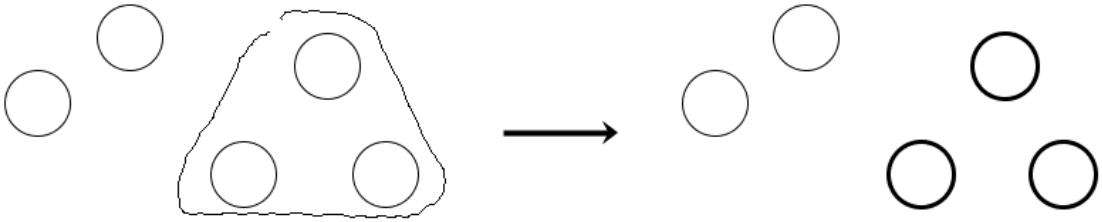
Creating, editing and deleting can be entirely done using a mouse, the keyboard only used to type in names of nodes and ports. We assign the left mouse button as the *drawing* button. The user only needs to draw two different shapes: a circle or a line; and the position and context of the shape will determine what intention the user had in mind.

Circle Shape

- A circle drawn on an empty part of the view that doesn't contain any elements will create a new node. Users can immediately type after creating a node to name it.

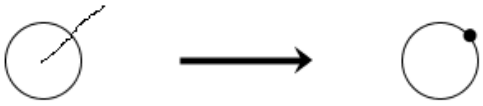


- A circle drawn that contains nodes will select those nodes so that the use can move them.

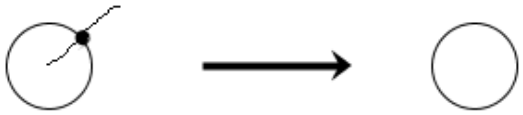


Line Shape

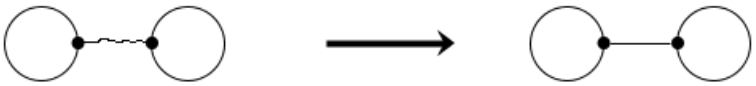
- A line drawn from inside a node to outside the node will create a port where the line intersects with the circumference of the node.



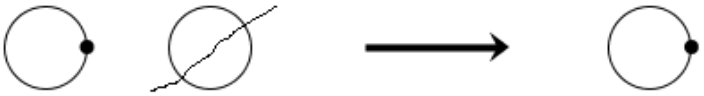
- A line drawn from an empty area outside a node into the node and through a port will delete the port.



- A line drawn from a port to a port will create an edge between both ports.



- A line starting and ending in an empty area will delete any nodes or lines it crossed with.



Automation

graphPaper automates or enhances certain interactions:

- Renaming a port will rename it on all nodes of the same type. The same concept applies to deleting nodes.
- Deleting nodes with edges connected to its ports will delete the edges automatically.
- Double-clicking on an empty space will create a node of the same type as the last created node. Doing the same on a node or port will allow the user to change its name.

Right Mouse Button

The right mouse button is used for moving either elements of the graph or moving the view. Right-clicking on a node or port will allow the user to move it. If a node is right-clicked that is part of a selected group, the entire group is moved. Right-clicking on an empty part of the view and then moving the mouse pushes the view in that direction. Zoom is also possible using a mouse wheel.

Rules

For rules, users create a node and name it an arrow node. This creates a box with the arrow node in the middle. They can then add all the nodes and edges on either side of this node to create the left hand side and right hand side, including interface rewiring. The box expands automatically to allow larger rules and edges automatically go through the arrow node if they connect a left hand side node to a right hand side node. Moving a node from one side to the other of the arrow node will update and change its edges to rewiring edges if needed. Figure 4.1 shows a mockup of a rule in graphPaper.

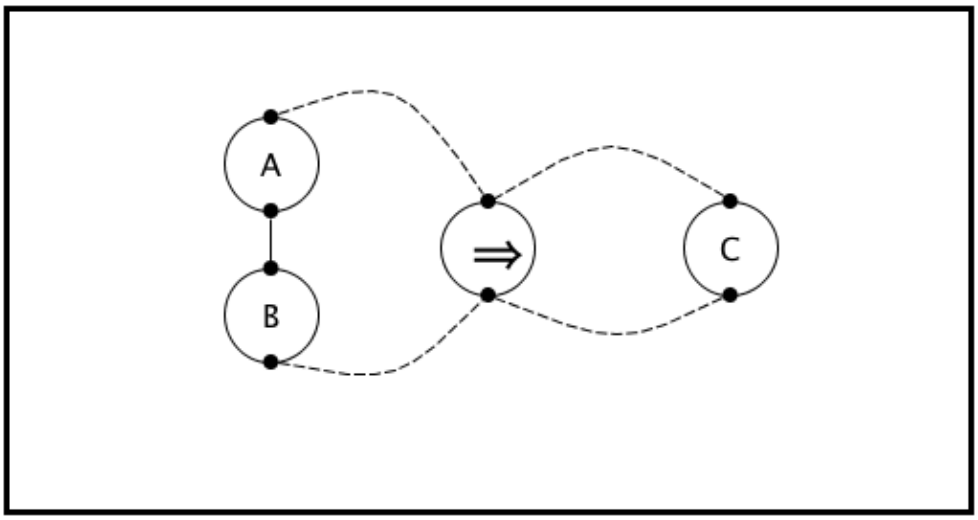


Figure 4.1: A rule in graphPaper.

4.2.2 Display

The method of interaction described above require a very minimal interface: icons for saving and loading and a set of icons for copy/pasting and undo/redoing. This provides a very clutter-free space to focus on creating graphs and rule.

Information is also displayed depending on context. While node names are always shown, port names only appear if the mouse cursor is near the port's node.

Colours are used to easily distinguish ports from their nodes and rewiring edges are drawn as dotted lines to differentiate them from normal edges.

Specific interaction net features are implemented that check the left hand side of a rule only contains two agents connected by their principal ports and that the interface is preserved.

4.3 Implementation and PORGY

graphPaper is currently being implemented in C++ using the SDL libraries[1] to ensure the tool is cross-platform. OpenGL[2][70] is used to draw the graphs efficiently and to enable effects like smooth zooming. Graphs and rules are saved in a graphPaper format and can also be exported in the TULIP graph format to be easily imported into PORGY as a starting graph and a collection of rules (this is done using the TULIP libraries). Information about the development of graphPaper can be found at [3].

[1] http://www.libsdl.org/
[2] http://www.opengl.org/

Chapter 5

Strategy Language

5.1 Introduction

We introduce the concept of graph program and give the syntax and semantics of the strategy language. In addition to the well-known constructs to select rewrite rules, the language provides primitives to focus on specific positions on the graph that are selected or banned for rewriting. Focusing is useful to program graph traversals, for instance, and is a distinctive feature of the language.

5.1.1 Located Graphs

Definition 5. *A* located graph G^Q_P *consists of a port graph* G *and two distinguished sets of nodes* P *and* Q *of* G*, called respectively the* position subgraph*, or simply position, and the* banned subgraph*.*

In a located graph G^Q_P, P represents the set of nodes of G where rewriting steps may take place (i.e., P is the focus of the rewriting) and Q represents the set of nodes of G where rewriting steps are forbidden. We give a precise definition below; the intuition is that subgraphs of G that overlap with P may be rewritten, if they are outside Q. When applying a port graph rewrite rule, not only the underlying graph G but also the position and banned subgraphs may change. A *located port graph rewrite rule*, defined below, specifies two disjoint sets of nodes M and N of the right-hand side R that are used to update the position and banned subgraphs, respectively. If M (resp. N) is not specified, R (resp. the empty graph $\emptyset$) is used as default. Since P and Q are sets of nodes of G, they can be considered as subgraphs of G*containing only nodes and no edges.* The same can be said of M and N. Below, the set operators union ($\cup$), intersection ($\cap$) and complement ($\setminus$, also known as *set difference*) apply to graphs considered as sets of nodes.

5.1.2 Located Port Graph Rewrite Rule

Definition 6. *A* located port graph rewrite rule *is given by a port graph rewrite rule* $L \Rightarrow R$ *and two disjoint subgraphs* M *and* N *of* R*. It is denoted* $(L \Rightarrow R)^N_M$*. We write* $G^Q_P \rightarrow^g_{(L\Rightarrow R)^N_M} {G'}^{Q'}_{P'}$ *and say that the located graph* G^Q_P *is rewritten to* ${G'}^{Q'}_{P'}$ *using* $(L \Rightarrow R)^N_M$ *at position* P *avoiding* Q*, if* $G \rightarrow_{L\Rightarrow R} G'$ *with a morphism* g *such that* $g(L) \cap P$ *is not empty and* $g(L) \cap Q$ *is empty; the new*

position P' is then defined as $P' = (P \setminus g(L)) \cup g(M)$; the new banned subgraph Q' is then defined as $Q' = Q \cup g(N)$.

In general, for a given rule $(L \Rightarrow R)^N_M$ and located graph G^Q_P, more than one morphism g, such that $g(L) \cap P$ is not empty and $g(L) \cap Q$ is empty, might exist (i.e., several rewriting steps at P avoiding Q might be possible). Thus, the application of the rule at the position P avoiding Q produces a *set of located graphs.*

5.1.3 Graph Programs

Definition 7. *A* graph program *is given by a set of located port graph rewrite rules $\mathcal{R}$, a strategy expression S (built from $\mathcal{R}$ using the grammar below) and a located graph G^Q_P. It is denoted by $\left[S_{\mathcal{R}}, G^Q_P\right]$, or simply $\left[S, G^Q_P\right]$ when $\mathcal{R}$ is clear from the context.*

The formal semantics of a graph program will be given below, using an Abstract Reduction System [72, 50, 25]. The idea is to build a set of rewrite derivations out of $\left[S, G^Q_P\right]$ according to the strategy S (i.e, a *derivation tree*).

A graph program $[S, G^Q_P]$ may define a non-terminating computation if there is an infinite derivation (written as $\perp$, an undefined result) that is a part of its result set. All terminating computations will produce *values* of the form $[\mathsf{Id}, G^Q_P]$ or $[\mathsf{Fail}, G^Q_P]$ (See Property 2 in Section 5.4).

Definition 8. *Given a graph program $[S, G^Q_P]$ and its derivation tree, the* result set *is the multiset of* values *($[\mathsf{Id}, G'^{Q'}_{P'}]$ or $[\mathsf{Fail}, G'^{Q'}_{P'}]$) in the tree, together with $\perp$ if there is an infinite branch. We say that a result set is a* success *if there exists at least one graph program $[\mathsf{Id}, G'^{Q'}_{P'}]$ (for some $G'^{Q'}_{P'}$) or is a* failure *otherwise.*

Definition 9. *A graph program $[S, G^Q_P]$ is strongly terminating if there are no infinite branches in its derivation tree. It is weakly terminating if there is at least one finite branch.*

Definition 10. *isSuccess() is a function that takes a graph program and returns:*

- *True: if the result set of the graph program contains at least one graph program of the type $[\mathsf{Id}, G^Q_P]$.*
- *False: if the result set of the graph program contains no graph programs of the type $[\mathsf{Id}, G^Q_P]$.*

The *isSuccess*() function is computable only if its input graph program is strongly terminating. In implementations, backtracking will be used to ensure that even if a strategy is weakly terminating, we will have computed a partial result set which only contains *values.*

5.2 Syntax

The syntax of the strategy language is given in Figure 5.1. The *strategy expressions* used in graph programs are generated by the non-terminal S. A strategy expression combines rule applications, generated by A, and focusing operations, generated by F. The application constructs and some

Let L, R be port graphs; F a set of nodes of G; M, N positions; ρ is a Property.

$$
\begin{array}{rrcl}
\textbf{(Focusing)} & F & := & \texttt{CrtGraph} \mid \texttt{CrtPos} \mid \texttt{CrtBan} \mid \texttt{AllSuc}(F) \mid \texttt{OneSuc}(F) \\
 & & \mid & \texttt{NextSuc}(F) \mid \texttt{Property}(\rho, F) \mid F \cup F \mid F \cap F \mid F \setminus F \mid \emptyset \\
\textbf{(Transformations)} & T & := & (L \Rightarrow R)^N_M \mid (T \parallel T) \\
\textbf{(Applications)} & A & := & \mathsf{Id} \mid \mathsf{Fail} \mid T \mid \texttt{one}(T) \\
\textbf{(Strategies)} & S & := & A \mid S; S \\
 & & \mid & \texttt{while}(S)\texttt{do}(S) \mid (S)\texttt{orelse}(S) \\
 & & \mid & \texttt{if}(S)\texttt{then}(S)\texttt{else}(S) \mid \texttt{isEmpty}(F) \\
 & & \mid & \texttt{setPos}(F) \mid \texttt{setBan}(F)
\end{array}
$$

Figure 5.1: Syntax of the strategy language.

Let $n \in \nabla_{\mathscr{N}}$, $attribute_1 \in \nabla_{\mathscr{A}}$, $attribute_2 \in \nabla_{\mathscr{A}} \cup \{*\}$, $port \in \nabla_{\mathscr{P}}$ and $value \in \nabla_{\mathscr{V}} \cup \{*\}$. (See Section 2.1.1)

$$
\begin{array}{rrcl}
\textbf{(Properties)} & \rho & := & n \\
 & & \mid & attribute_1 = value \\
 & & \mid & port.attribute_2 = value
\end{array}
$$

Figure 5.2: Syntax of the Property language.

of the strategy constructs in the language are strongly inspired from existing rewriting strategy languages such as Elan [24], Stratego [75] and Tom [17] and GP[67].

We describe the constructs informally first, and give their formal semantics in Section 5.3.

5.2.1 Focusing

These constructs are functions from graph programs to port graphs: they apply on a graph program $\left[S_{\mathcal{R}}, G^Q_P\right]$ and return a set of nodes of G. They are used in strategy expressions to change the positions P and Q where rules apply and to specify different types of graph traversals. In the following, F is a set of nodes of G.

- `CrtGraph` returns a set containing all the nodes in graph G.
- `CrtPos` returns a set equal to the current position subgraph P in the located graph.
- `CrtBan` returns a set equal to the current banned subgraph Q in the located graph.
- `AllSuc`(F) returns the set of nodes consisting of all the immediate successors of the nodes in

F, where an immediate successor of a node v is a node that has a port connected to a port of v.

- `OneSuc`(F) returns a set of nodes consisting of one immediate successor of a node in F, chosen non-deterministically.
- `NextSuc`(F) computes successors of nodes in F using for each node only a subset of its ports; we call the ports in this distinguished subset the *next* ports (so `NextSuc`(F) returns a subset of the nodes in `AllSuc`(F)).
- `Property`(ρ, F) is a filtering construct, that returns a set of nodes of G containing only the nodes from F that satisfy the decidable property ρ. It typically tests a property on nodes or ports, for instance:
  - `Property(`Add,F`)` returns all the nodes of F labelled "Add".
  - `Property(`Principal.Active = True,F`)` returns all the nodes of F that have a port labelled "Principal" that has an attribute labelled "Active" that is set to the value "True".
  - `Property(`Colour = $*$,F`)` returns all the nodes of F that have an attribute named "Colour", regardless of its value.
  - `Property(`Auxiliary.$*$ = $*$,F`)` returns all the nodes of F that have a port labelled "Auxiliary", regardless of any attributes and values that port might have.

  We can then combine multiple `Property(,)` operators with a $\cap$ (see below) to filter multiple times. For example `Property(`Mult,F`)` $\cap$ `Property(`Aux.Active = True,F`)` returns all the nodes in F that are labelled "Mult" *and* that have a port labelled "Aux" that has an attribute labelled "Active" that is set to the value "True".
- $\cup$, $\cap$ and $\setminus$ are the standard set theory operators that compute new expressions from the previous constructs.
- $\emptyset$ returns the empty set.

5.2.2 Transformations

The focusing subgraphs P and Q in the target graph and the distinguished graphs M and N in a located port graph rewrite rule are original features of the language. A rule can only be applied if at least a part of the redex is in P and cannot be applied on Q.

- $(L \Rightarrow R)^N_M$ represents the application of the rule $L \Rightarrow R$ at the current position P and avoiding Q in G^Q_P. $(L \Rightarrow R)^N_M$ returns Id if the rule can be applied and Fail if it can't.
- $T \parallel T'$ represents simultaneous application of T and T' on disjoint subgraphs of G and returns Id only if both applications are possible *simultaneously* and Fail otherwise.

5.2.3 Applications

- Id is a basic strategy that never fails and leaves the graph unchanged.
- Fail is also a basic strategy but leaves the graph unchanged and returns failure.
- T will compute all the possible applications of the transformation on the located graph, creating a new located graph for each application. In the derivation tree, this will create as many children as there are possible applications.
- `one(`T`)` will non-deterministically compute only one of the possible applications of the transformation and ignore the others.

5.2.4 Strategies

In this section, we describe the constructs used to build strategies.

- The expression $S;S'$ represents sequential application of S followed by S', as usual.
- `while(`S`)do(`S'`)` keeps on sequentially applying S' to G_P^Q while the expression S rewrites to Id on a copy of G_P^Q. If S returns Fail, then Id is returned.
- (S)`orelse(`S'`)` applies S if possible, otherwise applies S' and returns Fail if both S and S' fail.
- `if(`S`)then(`S'`)else(`S''`)` checks if the application of S on (a copy of) G_P^Q returns Id, in which case S' is applied to (the original) G_P^Q, otherwise S'' is applied to the original G_P^Q.
- `isEmpty(`F`)` returns Id if F returns an empty graph and Fail otherwise. This can be used for instance inside the condition of an `if` or `while`, to check if the strategy makes P empty or not.
- `setPos(`F`)` takes the graph resulting from a focusing expression and sets the current position subgraph P to that graph. `setPos()` always returns Id.
- `setBan(`F`)` takes the graph resulting from a focusing expression and sets the current banned subgraph Q to that graph. `setBan()` always returns Id.

5.3 Semantics

Each syntactic group (Focusing, Transformations, Applications and Strategies) requires semantics. The Focusing operators require functions to compute graphs (see Section 5.3.1) while the other syntactic groups use small step semantics on configurations in the form of rules (see Section 5.3.2).

5.3.1 Focusing Operators

The focusing operators defined by the grammar for F in Fig. 5.1 have a functional semantics. They apply to the current located graph, and compute a set of nodes G' (i.e., they return a set of nodes of G). For instance, `CrtGraph` applied to G_P^Q returns a set containing all nodes of G.

We define below the result of each focusing operation on a given located graph.

$\texttt{CrtGraph}(G_P^Q)$	$=$	G	$\texttt{CrtPos}(G_P^Q) = P \quad \texttt{CrtBan}(G_P^Q) = Q$
$\texttt{AllSuc}(F)(G_P^Q)$	$=$	G'	where G' consists of all immediate successors of nodes in F.
$\texttt{OneSuc}(F)(G_P^Q)$	$=$	G'	where G' consists of one immediate successor of a node in F, chosen non-deterministically.
$\texttt{NextSuc}(F)(G_P^Q)$	$=$	G'	where G' consists of the immediate successors, via ports labelled "next", of nodes in F.
$\texttt{Property}(\rho, F)(G_P^Q)$	$=$	G'	where G' consists of all nodes in F satisfying ρ.

If `AllSuc()` or `OneSuc()` have no immediate successors, `NextSuc()` is not given any nodes with ports labelled *next*, or `Property(,)` is given a set of nodes that don't satisfy ρ, then the empty set $\emptyset$ is returned.

We have three cases for ρ:

- n (where $n \in \nabla_{\mathscr{N}}$)
  that returns all the nodes in F that are labelled n.

- $attribute_1 = value$ (where $attribute_1 \in \nabla_{\mathscr{A}}$ and $value \in \nabla_{\mathscr{V}} \cup \{*\}$)
  that returns all the nodes in F that have $attribute_1$ as an attribute and where its value is equal to $value$. If $value$ is $*$ then it will return all nodes that have $attribute_1$ as an attribute regardless of its value.

- $port.attribute_2 = value$ (where $port \in \nabla_{\mathscr{P}}$, $attribute_2 \in \nabla_{\mathscr{A}} \cup \{*\}$ and $value \in \nabla_{\mathscr{V}} \cup \{*\}$)
  that returns all the nodes in F that have a port labelled $port$ and that port has an attribute labelled $attribute_2$ that has the value $value$. If $value$ is $*$ then the value is ignored. If both $attribute_2$ and $value$ are $*$ then they are both ignored and the nodes that are returned are all the nodes that have a port labelled $port$.

5.3.2 Transformation, Applications and Strategy Operators

Small Step Semantics and Configurations

The constructs in the grammars for T, A and S in Fig. 5.1 are defined by semantic rules given below, which are applied to configurations containing graph programs $\left[S, G_P^Q\right]$ (see Definition 7). We follow a *small step* style of operational semantics [65]. The semantic rules define computation steps on configurations.

A configuration is a multi-set $\{O_1, \ldots, O_n\}$ where O_i is a graph program or an intermediate object (denoted by angular brackets and that uses auxiliary operators). In the semantic rules, we abuse notation and identify a singleton multi-set with its element and work modulo the flattening of multi-sets and use the $\underset{flat}{=}$ symbol to represent a flattening, for instance $\{\{O_a, \ldots, O_b\}, O_y, \ldots, O_z\} \underset{flat}{=} \{O_a, \ldots, O_b, O_y, \ldots, O_z\}$.

The computation steps defined by the semantic rules can be seen as creating a *derivation tree* where each computation creates a branch in the tree. At any point during the computation, the

current configuration represents the set of all leaves of that tree. An example of a tree can be seen in Figure 5.3 and throughout Appendix A and B.

There is no need for a strategy to control the semantic rules since they are applied exhaustively.

Auxiliary Operators and Intermediate Objects

There exists three auxiliary operators:

- $;_2$
- $\texttt{if}_2()\texttt{then}()\texttt{else}()$
- $()\texttt{orelse}_2()$

These auxiliary operators are used to express intermediate steps for the ;, `if`()`then`()`else`() and ()`orelse`() operators respectively. To show their necessity, let us look at sequential application with the ; operator. If we have a graph program $[S_1; S_2, G_P^Q]$ we need to first apply S_1 to G_P^Q but also keep S_2 in memory to apply later (if the application of S_1 is successful). It is not possible to represent this with the current notion of graph programs so we use an intermediate object (using angular brackets) and the $;_2$ auxiliary operator to get $\langle [S_1, G_P^Q];_2 S_2, G_P^Q \rangle$. The left hand side of the operator being the next graph program we would like to compute and the right hand side serving as memory.

The same is done with $\texttt{if}_2()\texttt{then}()\texttt{else}()$ to create an intermediate object where we can compute the result of the *isSuccess*() funtion applied to the graph program that is the condition of the `if`()`then`()`else`().

The $()\texttt{orelse}_2()$ is used much like $;_2$, where its left hand side holds a graph program we would like to have evalauted and the right hand side storing a strategy in memory.

There are three types of semantic rules (see below):

- Four that take a *graph program* and return an *intermediate object*. $[] \rightarrow <>$
- Five that take an *intermediate object* and return a *set of graph programs*. $<> \rightarrow \{[], \ldots, []\}$
- The rest take a *graph program* and return a *set of graph programs*. $[] \rightarrow \{[], \ldots, []\}$

Type Variables

We type variables in rules by naming them as the initial symbol of the corresponding grammar with an index number if needed (for example: A_1 is a variable of application type; S_3 represents a strategy expression; F_2 represents a focusing expression).

Semantic Rules

- Graph rewrite rules are themselves strategy operators:

$$
\begin{array}{lcll}
[(L \Rightarrow R)_M^N, G_P^Q] & \rightarrow & \{[\mathsf{Id}, {G_1}_{P_1}^{Q_1}], \ldots, [\mathsf{Id}, {G_k}_{P_k}^{Q_k}]\} & \\
 & & \text{if } G_P^Q \rightarrow^{g_i}_{(L \Rightarrow R)_M^N} {G_i}_{P_i}^{Q_i} (\forall i, 1 \leq i \leq k) \text{ with } g_1 \ldots g_k \text{ pairwise different.} & \\
[(L \Rightarrow R)_M^N, G_P^Q] & \rightarrow & \{[\mathsf{Fail}, G_P^Q]\} & \text{if the rule is } \textit{not} \text{ applicable}
\end{array}
$$

- Parallelism is allowed through the operator || which works on applications only (not on general strategies). To define the semantics of $(L_1 \Rightarrow R_1)^{N_1}_{M_1} || \ldots || (L_k \Rightarrow R_k)^{N_k}_{M_k}$, we define a new rule $((L_1 \cup \cdots \cup L_k) \Rightarrow_{1\ldots k} (R_1 \cup \cdots \cup R_k))^{N_1 \cup \cdots \cup N_k}_{M_1 \cup \cdots \cup M_k}$ where $\Rightarrow_{1\ldots k}$ contains all the ports and edges of $\Rightarrow_i$ (for $1 \leq i \leq k$).

  It implements simultaneous application of rules at disjoint redexes.

$$
\begin{array}{lcll}
[(L_1 \Rightarrow R_1)^{N_1}_{M_1} || \ldots || (L_k \Rightarrow R_k)^{N_k}_{M_k}, G^Q_P] & \rightarrow & \{[((L_1 \cup \cdots \cup L_k) \Rightarrow_{1\ldots k} (R_1 \cup \cdots \cup R_k))^{N_1 \cup \cdots \cup N_k}_{M_1 \cup \cdots \cup M_k}, G^Q_P]\} & \\
 & & & \text{if } \forall i,\ L_i \cap P \neq \emptyset \\
[(L_1 \Rightarrow R_1)^{N_1}_{M_1} || \ldots || (L_k \Rightarrow R_k)^{N_k}_{M_k}, G^Q_P] & \rightarrow & \{[Fail, G^Q_P]\} & \text{otherwise}
\end{array}
$$

- The non-deterministic `one()` operator is defined with the following two rules:

$$
\begin{array}{lcll}
[\texttt{one}((L \Rightarrow R)^N_M), G^Q_P] & \rightarrow & \{[\mathsf{Id}, {G'}^{Q'}_{P'}]\} & \text{if } G^Q_P \rightarrow^g_{(L \Rightarrow R)^N_M} {G'}^{Q'}_{P'} \\
[\texttt{one}((L \Rightarrow R)^N_M), G^Q_P] & \rightarrow & \{[\mathsf{Fail}, G^Q_P]\} & \text{if the rule is } \textit{not} \text{ applicable}
\end{array}
$$

  While `one()` takes a transformation T as an argument in the syntax, the semantics will expect any parallel transformation to be reduced to a rule application first.

- Position definition:

$$
\begin{array}{lcll}
[\texttt{setPos}(F), G^Q_P] & \rightarrow & \{[\mathsf{Id}, G^Q_{P'}]\} & \text{where } P' \text{ is the result of } F \\
[\texttt{setBan}(F), G^Q_P] & \rightarrow & \{[\mathsf{Id}, G^{Q'}_P]\} & \text{where } Q' \text{ is the result of } F \\
 & & & \\
[\texttt{isEmpty}(F), G^Q_P] & \rightarrow & \{[\mathsf{Id}, G^Q_P]\} & \text{if } F \text{ is an empty graph} \\
[\texttt{isEmpty}(F), G^Q_P] & \rightarrow & \{[\mathsf{Fail}, G^Q_P]\} & \text{if } F \text{ is not an empty graph}
\end{array}
$$

- Sequential application: (where E is Id or Fail)

$$
\begin{array}{lcl}
[S_1; S_2, G^Q_P] & \rightarrow & \langle [S_1, G^Q_P];_2 S_2, G^Q_P \rangle \\
\langle \{[E_1, {G_1}^{Q_1}_{P_1}], \ldots, [E_k, {G_k}^{Q_k}_{P_k}]\};_2 S_2, G^Q_P \rangle & \rightarrow & \{[E_1; S_2, {G_1}^{Q_1}_{P_1}], \ldots, [E_k; S_2, {G_k}^{Q_k}_{P_k}]\} \\
[Id; S, G^Q_P] & \rightarrow & \{[S, G^Q_P]\} \\
[Fail; S, G^Q_P] & \rightarrow & \{[Fail, G^Q_P]\}
\end{array}
$$

  Here, we *promote* S_1 with the first rule so that it can be applied to G^Q_P. The next rule takes all the graph programs in the result set that is returned and puts each of them in sequence with $;_2 S_2$. The following two rules then deals with the outcomes of the concatenation done in the previous rule.

- Conditional :

$$
\begin{array}{lcl}
[\texttt{if}(S_1)\texttt{then}(S_2)\texttt{else}(S_3), G^Q_P] & \rightarrow & \langle \texttt{if}_2(isSuccess([S_1, G^Q_P]))\texttt{then}(S_2)\texttt{else}(S_3), G^Q_P \rangle \\
\langle \texttt{if}_2(True)\texttt{then}(S_2)\texttt{else}(S_3), G^Q_P \rangle & \rightarrow & \{[S_2, G^Q_P]\} \\
\langle \texttt{if}_2(False)\texttt{then}(S_2)\texttt{else}(S_3), G^Q_P \rangle & \rightarrow & \{[S_3, G^Q_P]\}
\end{array}
$$

  The first rule *promotes* S_1 which is the strategy we are trying to check so that it can be applied to G^Q_P and is put into the $isSuccess()$ function. Depending on the result of $isSuccess()$, S_2 (if $isSuccess()$ returns $True$) or S_3 (if $isSuccess()$ returns $False$) is applied to G^Q_P.

- Iteration:

$$[\texttt{while}(S_1)\texttt{do}(S_2), G^Q_P] \rightarrow \langle \texttt{if}_2(isSuccess([S_1, G^Q_P]))\texttt{then}(S_2; \texttt{while}(S_1)\texttt{do}(S_2))\texttt{else}(\mathsf{Id}), G^Q_P\rangle$$

  Iteration uses a recursive rule transforming a `while` into a `while` nested in an $\texttt{if}_2()\texttt{then}()\texttt{else}()$.

- Priority choice: (where E is Id or Fail)

$$\begin{aligned}
[(S_1)\texttt{orelse}(S_2), G^Q_P] &\rightarrow \langle([S_1, G^Q_P])\texttt{orelse}_2(S_2), G^Q_P\rangle \\
\langle(\{[E_1, G_1{}^{Q_1}_{P_1}], \ldots, [E_k, G_k{}^{Q_k}_{P_k}]\})\texttt{orelse}_2(S_2), G^Q_P\rangle &\rightarrow \{[E_1, G_1{}^{Q_1}_{P_1}], \ldots, [E_k, G_k{}^{Q_k}_{P_k}]\} \\
&\quad \text{if } isSuccess(\{[E_1, G_1{}^{Q_1}_{P_1}], \ldots, [E_k, G_k{}^{Q_k}_{P_k}]\}) = True \\
\langle(\{[E_1, G_1{}^{Q_1}_{P_1}], \ldots, [E_k, G_k{}^{Q_k}_{P_k}]\})\texttt{orelse}_2(S_2), G^Q_P\rangle &\rightarrow [S_2, G^Q_P] \\
&\quad \text{if } isSuccess(\{[E_1, G_1{}^{Q_1}_{P_1}], \ldots, [E_k, G_k{}^{Q_k}_{P_k}]\}) = False
\end{aligned}$$

  Here, S_1 is *promoted* so that it can be applied to G^Q_P. The next two rules return the result set (if $isSuccess()$ on the result set is successful) or S_2 otherwise.

  () `orelse` () is a primitive operator here since rewriting $(S_1)\texttt{orelse}(S_2)$ to $\texttt{if}(S_1)\texttt{then}(S_1)\texttt{else}(S_2)$ creates two instances of S_1 that could be successfully executed in the `if()` part of the `if()then()else()` but then create a failure when it is executed as the `then()` part. This is due to the non-determinism of the `OneSuc(a)`nd `one()` operators.

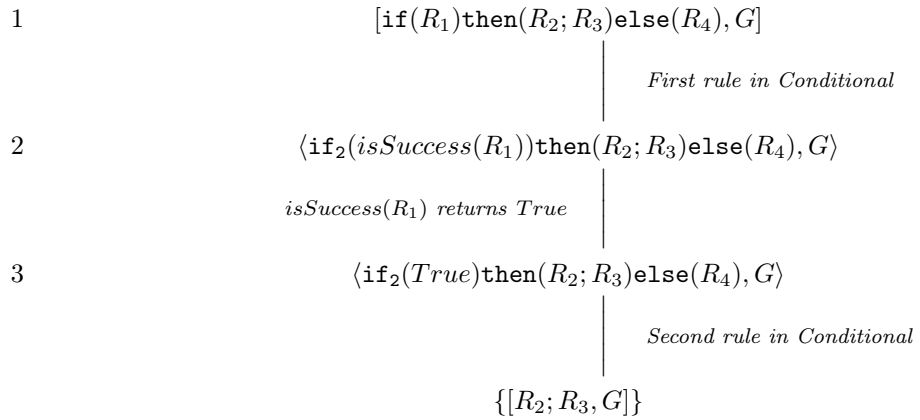
5.3.3 An Example

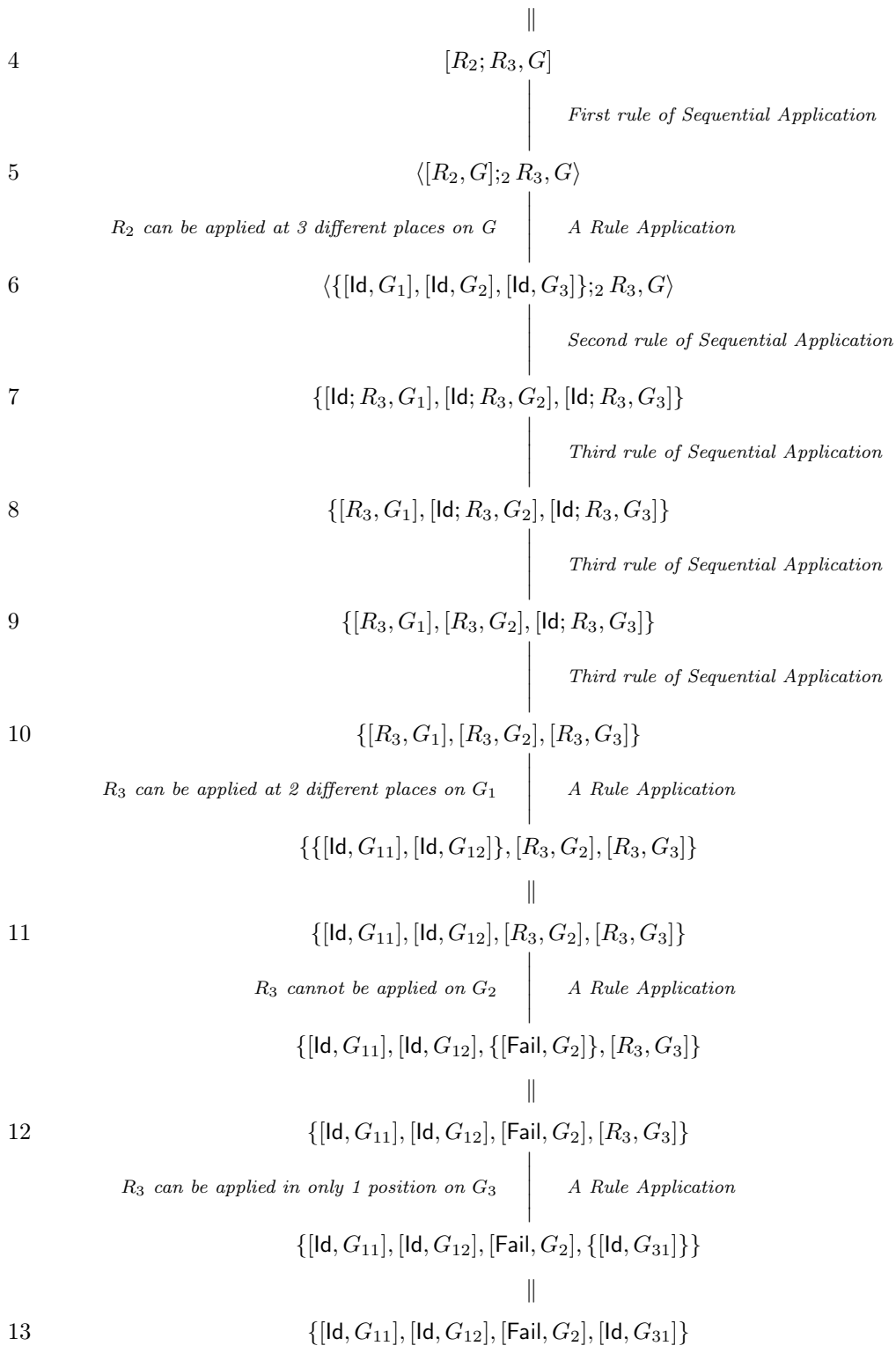
Let us consider the graph program $[\texttt{if}(R_1)\texttt{then}(R_2; R_3)\texttt{else}(R_4), G^Q_P]$ where R_1,R_2,R_3 and R_4 are rules and G a port graph.

We will leave the *position subgraph* and *banned subgraph* out of the notation for the sake of clarity. An application of a semantic rule is denoted by a long single line while a short double line is a multi-set flattening application. Since semantic rules are applied exhaustively and in any order, different derivations are possible. In a terminating graph program that doesn't use the non-deterministic operators `OneSuc()` and `one()`, we will end up with the same result set no matter what order the semantic rules are applied in (See Section 5.4).

For clarity, we will apply the semantic rules in the following example from left to right.

Using the semantics defined above we get the following derivation:





Step 13 is the final step since all graph programs in its set are irreducible. This is our result set. Below, we show the same graph program but in tree form (see Figure 5.3), where if a rule is applicable in more than one place in G the tree branches for each possibility. Each node in the tree represents a derivation step and the numbers below each node show the correspondence between the tree and the above semantic derivation. To see the partially constructed tree for a particular derivation (for example step 8), simply remove all nodes that have all their numbers strictly higher than 8 so that all nodes containing 8 are leaf nodes (see Figure 5.4).

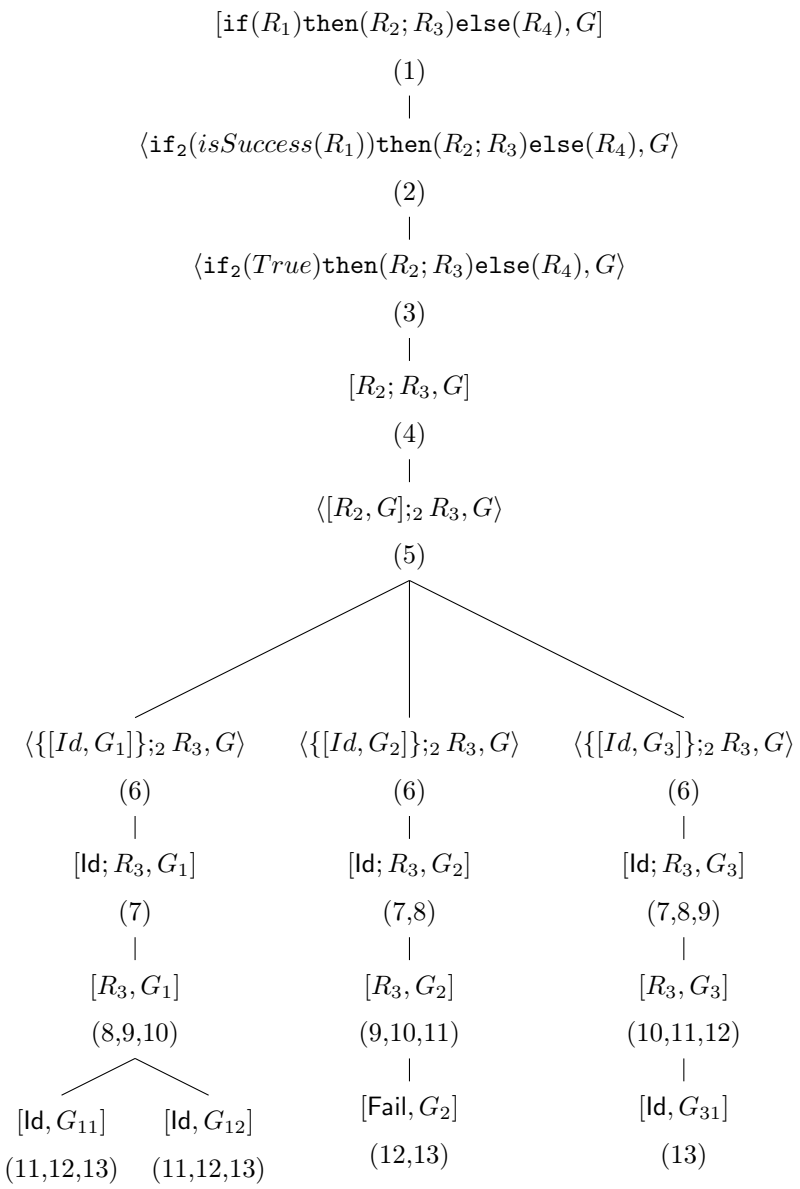


Figure 5.3: The fully developed derivation tree.

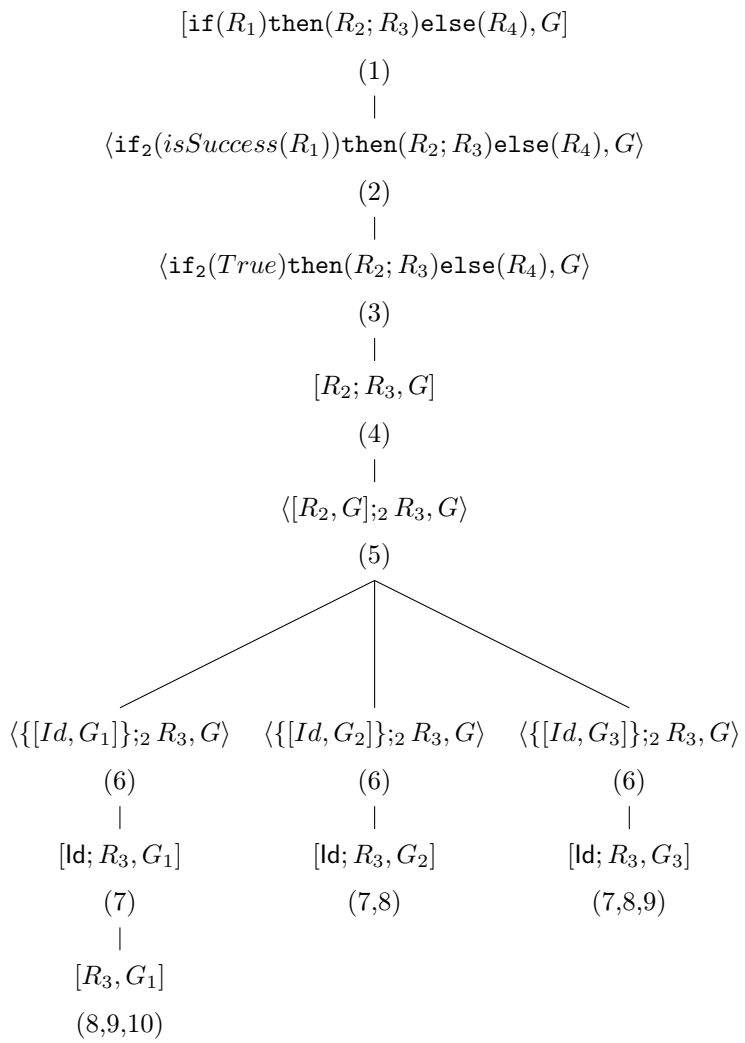


Figure 5.4: The derivation tree developed to step 8.

The tree notation here is interesting since the PORGY tool described in Chapter 7 uses that notation to display derivations. More details of this are given in that chapter.

5.4 Properties

5.4.1 Termination

Graph programs are not terminating in general, however we can characterise a terminating sublanguage and characterise the normal forms of terminating graph programs.

Property 1 (Termination)**.** *The sublanguage that excludes the* while *construct does not generate infinite derivations with the semantic rules.*

Proof. To prove that a graph program in this sublanguage does not generate an infinite derivation, we use an interpretation of the configurations used in the semantic rules and show that this interpretation strictly decreases with each application of a rewrite rule within the semantics. The interpretation of a configuration is defined as follows: $int(\{O_1, \ldots, O_n\}) = \sum_{i=1}^{n} \mathcal{I}(O_i)$ where $\mathcal{I}([S, G]) = size(S)$ and $\mathcal{I}(\langle S, G\rangle) = size(S)$. The size function is defined as:

$$size(\mathsf{Id}) = 0 \qquad size(\mathsf{Fail}) = 0 \qquad size((L \Rightarrow R)^N_M) = 1$$
$$size(\texttt{one}(T)) = 1$$
$$size(\texttt{setPos}(F)) = 1 \qquad size(\texttt{setBan}(F)) = 1 \qquad size(\texttt{isEmpty}(F)) = 1$$
$$size(S_1; S_2) = 2 + size(S_1) + size(S_2)$$
$$size(C ;_2 S) = 1 + int(C) + size(S)$$
$$size(\texttt{if}(S_1)\texttt{then}(S_2)\texttt{else}(S_3)) = 2 + size(S_1) + size(S_2) + size(S_3)$$
$$size(\texttt{if}_2(B)\texttt{then}(S_2)\texttt{else}(S_3)) = 1 + size(S_2) + size(S_3)$$
$$size((S_1)\texttt{orelse}(S_2)) = 2 + size(S_1) + size(S_2)$$
$$size((C)\texttt{orelse}_2(S_2)) = 1 + int(C) + size(S_2)$$
$$size(T_1||T_2) = 1 + size(T_1) + size(T_2)$$

If we calculate the interpretation for both sides of our semantic rules, we see that the left hand side is strictly greater than the right hand side, therefore showing that our interpretation strictly decreases with each application of a semantic rule.

□

5.4.2 Normal Forms

Property 2 (Characterisation of Normal Forms)**.** *A strongly terminating graph program* $\left[S, G^Q_P\right]$ *eventually rewrites using the semantic rules to a result set of the form* $\{A_1, \ldots, A_n\}$*, where each* A_i *is a* value*. A weakly terminating graph program eventually reduces to a configuration containing only values and reducible graph programs (at least one of each).*

Proof. By inspection of the semantic rules, every graph program $[S, G^Q_P]$ different from $[\mathsf{Id}, G^Q_P]$ or $[\mathsf{Fail}, G^Q_P]$ can be matched by a left-hand side of a rule. This is true because S has a top operator which is one of the syntactic constructions and there is a semantic rule which applies to it. Moreover every expression of the form $\langle X, G^Q_P\rangle$ is reducible, because X either contains a graph program $[S, G^Q_P]$ different from $[\mathsf{Id}, G^Q_P]$ or $[\mathsf{Fail}, G^Q_P]$ that so can be matched by a left-hand side of a rule as above, or one of the auxiliary rules can apply.

□

This implies that the computation arising from a graph program cannot be blocked before reaching a configuration containing a value, unless the if-part of a conditional is a non-terminating program, in which case the *isSuccess*() function (which is non computable) will block the computation.

The language contains non-deterministic operators in some of its syntactic categories: `OneSuc()` for Focusing and `one()` for Applications.

5.4.3 Result Set

Property 3 (Result Set)**.** *Each graph program in the sublanguage that excludes* `OneSuc()` *and* `one()`*, has at most one* result set*.*

Proof. Consequence of the fact that the semantic rules form an orthogonal system: they are left-linear and the conditions guarantee non-superposition, see [51].

□

Property 4 (Result Set with `one()`)**.** *A graph program in the sublanguage that excludes* `OneSuc()` *(but contains* `one()`*) produces a result set that is a (non-strict) subset of the same graph program where all occurrences of* `one()` *have been removed from its strategy.*

Proof. Assuming a transformation T is applicable, the application of T produces a result set $\{[\mathsf{Id}, {G_1}^{Q_1}_{P_1}], \ldots, [\mathsf{Id}, {G_k}^{Q_k}_{P_k}]\}$ while the application of $\texttt{one}(T)$ produces a result set with a single element: $\{[\mathsf{Id}, {G_i}^{Q_i}_{P_i}]\}$, where $\{[\mathsf{Id}, {G_i}^{Q_i}_{P_i}]\} \subseteq \{[\mathsf{Id}, {G_1}^{Q_1}_{P_1}], \ldots, [\mathsf{Id}, {G_k}^{Q_k}_{P_k}]\}$. The result set of a graph program $[\texttt{one}(rule_1), G^Q_P]$ is therefore a subset of the result set of $[rule_1, G^Q_P]$. The result follows by induction. □

Intuitively, in a strategy, an application of a transformation with `one()` creates a configuration that is a subset of the configuration computed by the same transformation. Successive applications of transformations with `one()` generate configurations that are successive subsets of the same configurations where the transformations were not applied with `one()`.

5.4.4 Completeness

Property 5 (Completeness)**.** *The set of graph programs* $\left[S, G^Q_P\right]$ *is Turing complete.*

Proof. Consequence of Theorem 1 in [47], which shows that sequence, iteration and the ability to apply a rule from a set is sufficient to simulate a Turing machine. Iteration (`while`) and sequence (;) are primitives in our language. The application of a rule from a set can be defined using the `orelse` operator. □

It is also interesting to consider which sublanguages of our language are Turing complete.

Property 6 (Complete sublanguage)**.** *The sublanguage consisting of graph programs where* $S_{\mathcal{R}}$ *is built from* Id, Fail, *rules* $(L \Rightarrow R)^N_M$ *in* $\mathcal{R}$*, sequential composition (;), iteration (*while*), and* `orelse` *is Turing complete.*

Proof. Since Turing machine computations can be simulated using ;, while and `orelse`, the result follows.

□

The same result could be obtained by replacing `orelse` with the conditional construct if then else. Perhaps more surprising is the fact that we can simulate Turing machine computations using the sublanguage consisting of focusing operators and sequential composition (;), iteration (`while`) and rule application $(L \Rightarrow R)^N_M$ together with Id, Fail, `setPos()` and `setBan()`. This result follows from [33], where it is proved that the computations of any Turing machine can be simulated by a term rewriting system consisting of just one rule, using a strategy that forces the reduction steps to take place at specific positions (called S-deep-only derivations in [33]). We omit the details of the proof, but highlight the fact that, given a sequence of transitions in the Turing machine,

Dauchet [33] shows how to build a rewrite rule to simulate the transitions, using a strategy to select the position for rewriting according to the instruction used by the machine. The `setPos()` construct can be used to simulate Dauchet's strategy directly, by moving the focus of rewriting to the corresponding subterm after each rewrite step.

Chapter 6

Application of the Strategy Language

6.1 Non-Primitive Operators

In this section we give examples to illustrate the expressivity of the language. The `not` and `try` operators, well-known in strategy languages for term rewriting, are not primitive operators in our language but can also be derived as well as the `repeat`(S) construct that applies the strategy S as long as possible. We can also define bounded iteration and a for-loop; $|||$ is a weaker version of $\|$.

- `not`$(S) \triangleq$ `if`(S)`then`(Fail)`else`(Id) fails if S succeeds and succeeds if S fails; the graph is unchanged.
- `try`$(S) \triangleq (S)$`orelse`(Id) is a strategy that behaves like S if S succeeds, but if S fails then it behaves like Id.
- `repeat`$(S) \triangleq$ `while`(S)`do`(S) applies S as long as possible.
- `while`(S_1)`do`(S_2)`max`$(n) \triangleq$ `if`(S_1)`then`$(S_2;$`if`(S_1)`then`$(S_2; \ldots)$`else`(Id)`else`(Id) representing a series of `if()`s of the same form, such that there are exactly n occurrences of S_2.
- `for`(n)`do`$(S) \triangleq S; \ldots; S$ where S is repeated n times.
- $A \,|||\, A'$ is similar to $A||A'$ except that it returns Id if at least one application of A or A' is possible.

  $$A_1|||A_2 \triangleq \texttt{if}(A_1)\texttt{then}(\texttt{if}(A_1||A_2)\texttt{then}(A_1||A_2)\texttt{else}(A_1))\texttt{else}(A_2)$$

  This operator can be generalised to n applications in parallel.

6.2 Traversals

Using focusing (specifically the `Property` construct) we can create concise strategies that perform traversals. In this way, we can define outermost or innermost term rewriting (on trees) without needing to change the rewrite rules. This is standard in term-based languages such as ELAN[24] or Stratego[75][26]; here we can also define traversals in graphs.

6.2.1 Outermost Traversal

Outermost rewriting on trees: we begin by defining the abbreviations $start \triangleq \texttt{Property}(root, \texttt{CrtGraph})$, which selects the subgraph containing just the root of the tree. If we define the *next* ports (see definition of the $\texttt{NextSuc}(F)$ operator in Section 5.2.1) for each node in the tree to be the ones that connect with their children, then the strategy for outermost rewriting with a rule R is:

$\texttt{setPos}(start);$
$\texttt{while}(\texttt{not}(\texttt{isEmpty}(\texttt{CrtPos})))\texttt{do}$
$(\texttt{if}(R)\texttt{then}(R; \texttt{setPos}(start))\texttt{else}(\texttt{setPos}(\texttt{NextSuc}(\texttt{CrtPos}))))$

In this strategy, if R can be applied then apply it and set the position back to the root of the tree. If R cannot be applied $\texttt{setPos}(\texttt{NextSuc}(\texttt{CrtPos}))$ will make all the children of all the elements in the current position the new current position, thus descending one step into the tree.

6.2.2 Innermost Traversal

Innermost rewriting on trees: using banned subgraphs, this can be written by using $start \triangleq \texttt{Property}(position = leaf, \texttt{CrtGraph})$ that selects the leaves of the tree (by looking at a node attribute labelled *position* and checking if the value *leaf* is assigned to it), and $rest \triangleq \texttt{CrtGraph} \setminus start$ by defining, for each node, the *next* port to be the one that connects with the parent node:

$\texttt{setPos}(start); \texttt{setBan}(rest);$
$\texttt{while}(\texttt{not}(\texttt{isEmpty}(\texttt{CrtPos})))\texttt{do}($
$\quad \texttt{if}(R)$
$\quad \texttt{then}(R; \texttt{setPos}(start); \texttt{setBan}(rest))$
$\quad \texttt{else}(\texttt{setPos}(\texttt{NextSuc}(\texttt{CrtPos})); \texttt{setBan}(\texttt{CrtBan} \setminus \texttt{CrtPos}))$
$)$

Here, if R can be applied then apply it and set the position back to the leaves of the tree and put all the other elements of the tree in to the banned subgraph. If R cannot be applied, the position travels up the tree by one level with $\texttt{setPos}(\texttt{NextSuc}(\texttt{CrtPos}))$ and the banned subgraph is updated again to all the remaining elements of the tree (with $\texttt{setBan}(\texttt{CrtBan} \setminus \texttt{CrtPos})$).

6.2.3 Interface Normal Form Traversal

Interface Normal Form: this is a generalisation of the outermost strategy for graphs (not necessarily trees), used to evaluate interaction nets (see Section 2.1.2 for more details). The *interface* of a port graph G is the set of nodes of G that have a free port. They can be selected by defining the position $Int \triangleq \texttt{Property}(position = interface, G)$ where *position* is an attribute for all nodes that is assigned a value of *interface* if the node belongs to the interface. The idea is to rewrite as near as possible to the interface (i.e., outermost). For each port node, we define its *next* port to be the principal port of the node and compute Interface Normal Form with respect to the rule R using a strategy similar to the outermost one, replacing *start* with *Int*.

$\texttt{setPos}(Int);$
$\texttt{while}(\texttt{not}(\texttt{isEmpty}(\texttt{CrtPos})))\texttt{do}($

```
if(R)then(R; setPos(Int))else(
    if(isEmpty((CrtPos ∪ NextSuc(CrtPos)) \ CrtPos))
    then(setPos(∅))
    else(setPos(CrtPos ∪ NextSuc(CrtPos)))
)
)
```

For this strategy, if R can be applied then apply it and reset the position to the interface. If R is not applicable, we need to find all the *next* elements of the interface and add them to the position subgraph but we also need to take into account cycles. If there is a cycle in the graph then eventually $\texttt{CrtPos} \cup \texttt{NextSuc}(\texttt{CrtPos})$ will return a CrtPos that won't have changed and this would push the strategy into an infinite loop. To prevent this we check if $(\texttt{CrtPos} \cup \texttt{NextSuc}(\texttt{CrtPos})) \setminus \texttt{CrtPos}$ is empty or not. If it is empty then we have found a cycle so we set CrtPos to $\emptyset$ so that the strategy exits and terminates. If it isn't empty then we set the current position to $\texttt{CrtPos} \cup \texttt{NextSuc}(\texttt{CrtPos})$ and loop to try apply R in this updated position subgraph.

6.3 Arithmetic with Interaction Nets

In a term rewriting system with a finite signature natural numbers are often represented using two function symbols S and 0. Then the number n is represented by a term $S(S(\ldots S(0)\ldots)$ with n occurrences of S (as described previously in Section 2.1.2). This representation is inefficient, but in [55] it is shown that using interaction nets we can implement efficiently arithmetic operations on integers, with a finite signature, by representing a number z in the form of a difference list $p-q$. The I agent is the *head* of a number and holds two lists of S agents: a left list containing p and a right list containing q. See Figure 6.1 for an example of the number 1 represented as $4-3$. We also note that there are infinite representations for each number in this way: $1 = 4-3 = 6-5 = 7-6 = ...$

The *open* rule extracts both lists from a number so that they can be used for arithmetic operations. If two lists are put head to head, the *reduce* rule will eventually return a single list containing the absolute value of the difference of the lists. The *negate* rule, switches the left and right list of a number, giving us its negative.

Using these three rules we can model *Addition*, *Negation* and *Subtraction*, as seen in Figure 6.2

If we liken the size of a graph to memory space, we could then like to prioritise the *reduce* rule so that the graph is always kept at its smallest:

```
repeat(repeat(reduce);try((negate) orelse (open)))
```

The strategy will apply *reduce* as many times as possible and then attempt to apply either *negate* or *open* before looping back to apply *reduce* again as many times as possible.

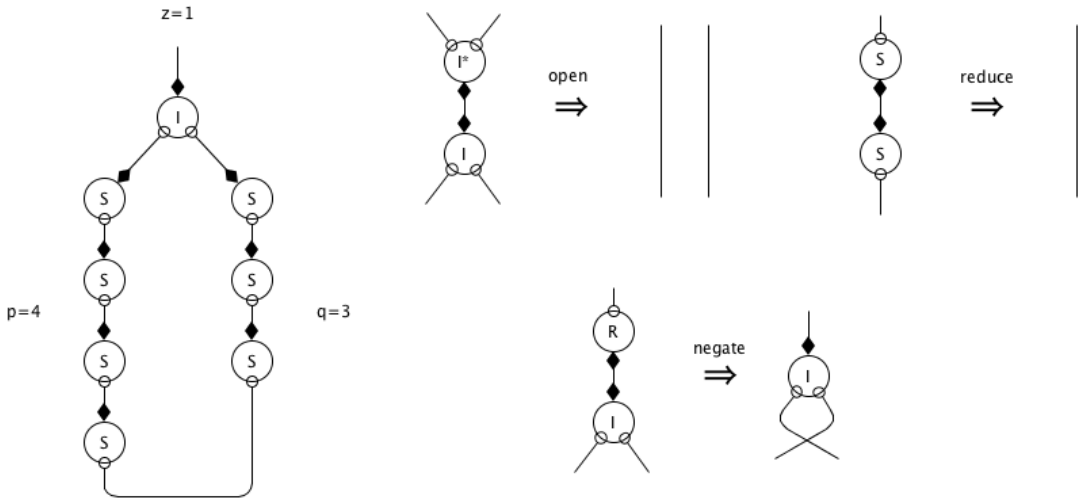


Figure 6.1: An example number and the *open*, *reduce* and *negate* rules.

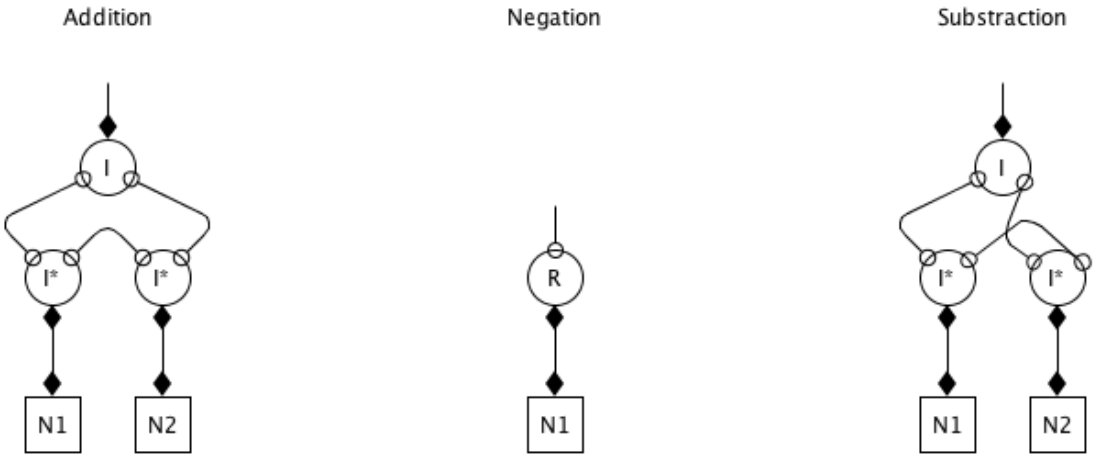


Figure 6.2: Modelling Addition, Negation and Subtraction.

6.4 Von Koch Fractals using Port Graphs

To draw a Von Koch Fractal (see Figure 6.3), we only need one node type and one rule. Our initial graph is a triangle and has one (and only one) of the nodes in P. We define a rule *vonKoch* of type $(L \Rightarrow R)^N_M$ (see *VFK* in Figure 6.3) such that M contains the right-most node from the right-hand side of the rule.

This means that after each application of *vonKoch*, the following segment will be the only application possible due to P. Our rule will then *travel* round the triangle segment by segment gradually creating a more complex fractal after each round trip.

In Figure 6.4, we can see three successive applications of *vonKoch*. Nodes drawn with dashed lines are nodes that are in P. The VKF strategy used is:

`repeat(vonKoch)`

To do just n iterations, we can use the following strategy:

`for(n)do(vonKoch)`

Without the notion of position in the strategy language, the application of the VKF rule would have been random and the fractal being generated will not necessarily be balanced. The strategy language with its use of focusing constructs, allows the rewriting to go round the triangle creating the fractal in a concise way.

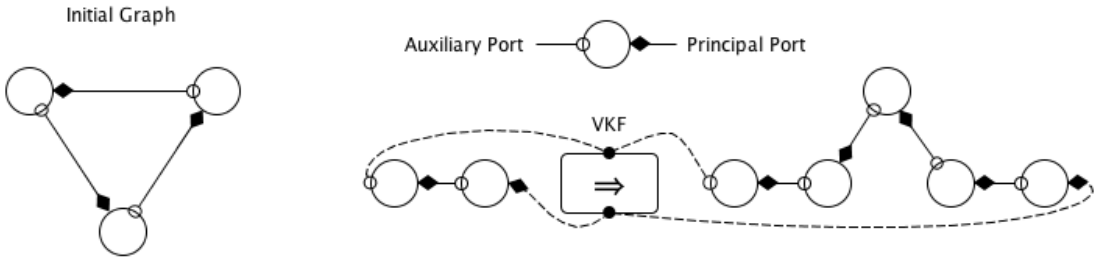


Figure 6.3: Modelling the Von Koch Fractal.

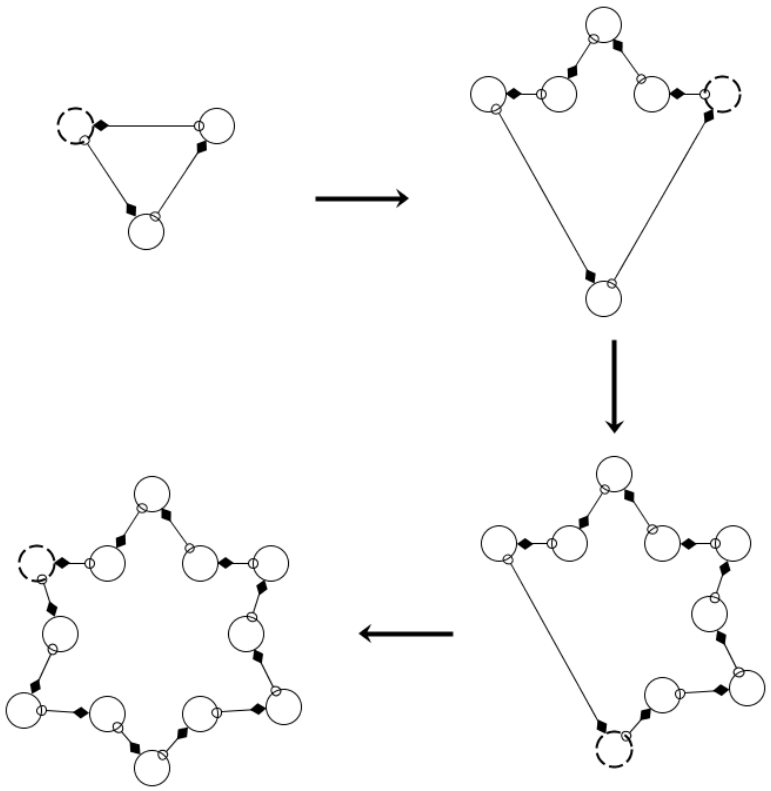


Figure 6.4: The Von Koch Fractal.

6.5 Sierpinski's Triangle Generation Using Port Graphs

Sierpinski's Triangle is a fractal that can be recursively defined [62]. An example of its evolution can be seen in Figure 6.5.

Figure 6.5: The evolution of a Sierpinski Triangle.

For this example, we only need one type of node so an empty-labelled node will be used to keep the generated graphs simple. This node will have four ports with identical names since the port being used here doesn't affect the system we are modelling, we again will not explicitly draw the ports to simplify the drawing.

The base case for Sierpinski's Triangle (going from one triangle to three) can be used as the only rule needed to generate iterations of the fractal and can be seen in Figure 6.6. In this rule, we have drawn the elements to put into the subgraph N of the rule with dotted lines for clarity (these nodes will be added to the *banned subgraph* Q during the rewriting step). All other nodes in the right hand side are put into M subgraph (to be added to the *position subgraph* P during the rewrite step).

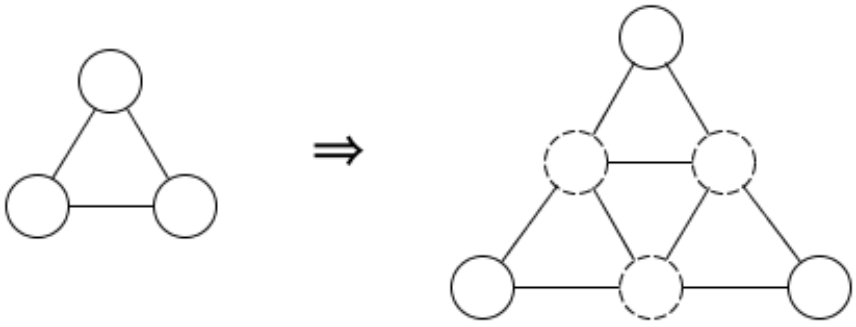


Figure 6.6: The only Sierpinski Triangle rule.

The starting graph will contain a simple triangle (like the one in the left hand side of the rule) with all its elements in the graph's position subgraph P. We then try to apply the rule as many times as possible. When the rule is no longer applicable we have created one full iteration of the fractal. To proceed to the next iteration we just need to take all the nodes that aren't banned ($G \setminus Q$) and add them into P and then empty Q. We then apply the rule as many times as possible and that will result in the next iteration.

A strategy for creating n iterations of Sierpinski's Triangle would be:

$$\texttt{for}(n)\texttt{do}(\texttt{setPos}(\texttt{CrtGraph} \setminus \texttt{CrtBan}); \texttt{setBan}(\emptyset); \texttt{repeat}(\mathit{rule}))$$

where *rule* is the rule defined in Figure 6.6.

Here, the tactical use of *banning* and *position* ensures that only the correct triangles are recursed on, and they are only each used once per iteration.

The following three figures (6.7, 6.8 and 6.9) each show an iteration of the execution of the

above strategy (With the final one only showing the `setPos(CrtGraph \ CrtBan)`). In the figures, nodes labelled P are in the P subgraph and black nodes denotes nodes in the Q subgraph.

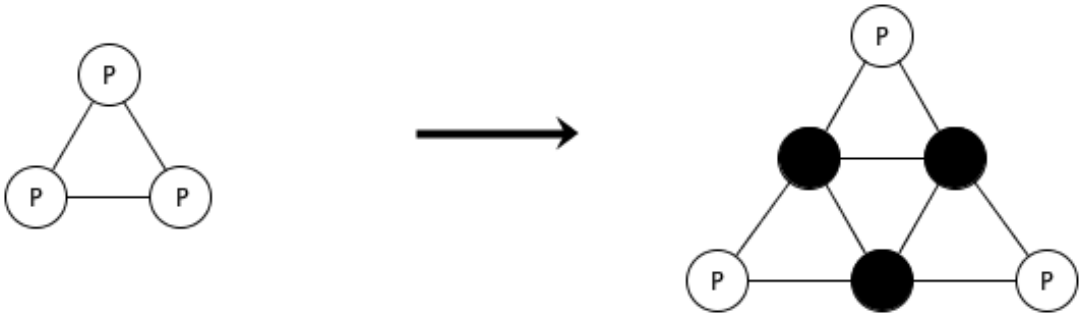


Figure 6.7: The *first* derivation.

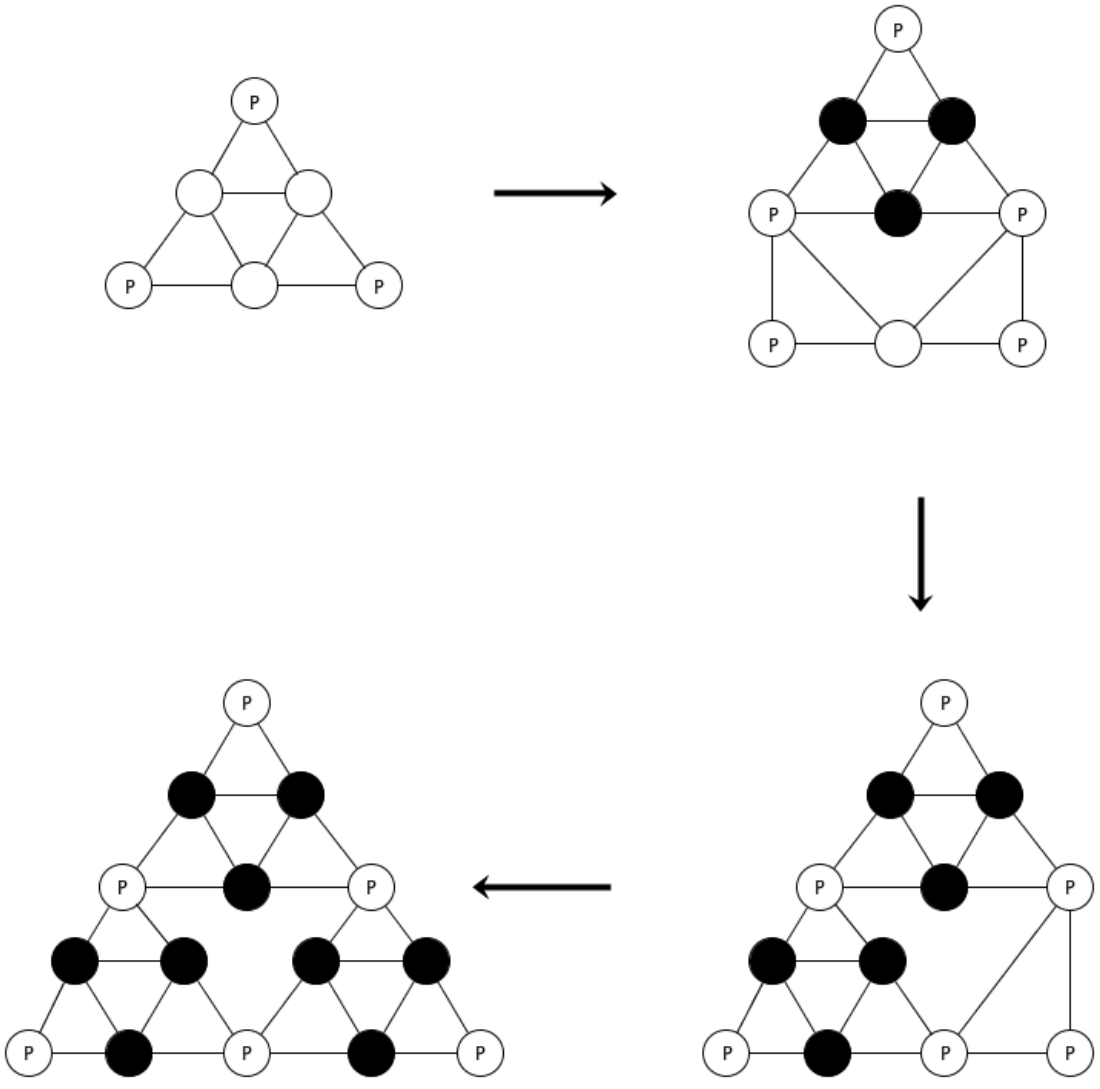


Figure 6.8: The *second* derivation.

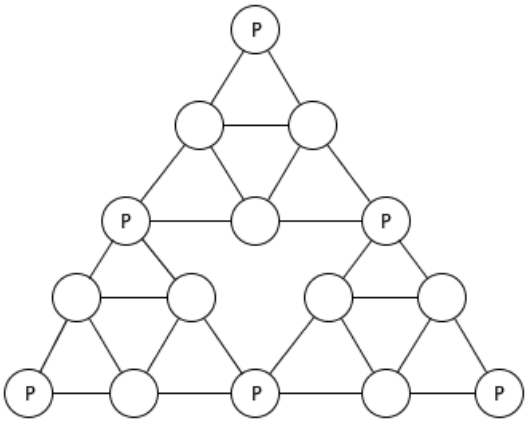


Figure 6.9: The start of the *third* derivation.

The `repeat(rule)` terminates since *rule* puts elements into the *banned subgraph* in a way that gradually isolated the remaining elements in P, eventually making *rule* inapplicable.

6.6 Game and AI Example Using Pacman

To simulate a game of pac-man, we use the initial graph in Figure 6.10 with the five types of nodes depicted. We assume all nodes in this system apart from the *space* node to have one port each, used to connect to a *space* node. The *space* node has 6 ports, one for each direction (up, down, left and right), one for pac-man and ghosts to connect to (the *character* port, depicted by an arrowhead) and one for pac-dots (the *dot* port, depicted by a filled black circle). For visual simplicity, we do not draw any free ports or ports whose state does not affect a rewrite rule. So conversely, if a port is explicitly drawn in a rule, it must be free.

The rewrite rules for pac-man can be found in Figure 6.11. Each rule in this figure is a *macro* rule: for the *explore* rule, we don't say which directional ports are used to connect the two *space* nodes and in fact would have to create four version of the rule, one for each direction possible between two *space* nodes. For simplicity's sake Figure 6.11 uses these *macro* rules but for an implementation all the explicit rules need to be created (unless the implementation has a macro feature coded, something in PORGY's future work).

These rules, with the help of a strategy, simulate a basic "artificial intelligence" for pac-man. Pac-man's first instinct is to flee any nearby ghosts (rules *flee1* and *flee2*). If pac-man is not near any ghosts, he then seeks out pac-dots (rule *getPacDot*) and then if not near any pac-dots, he proceeds to explore the level (rule *explore*).

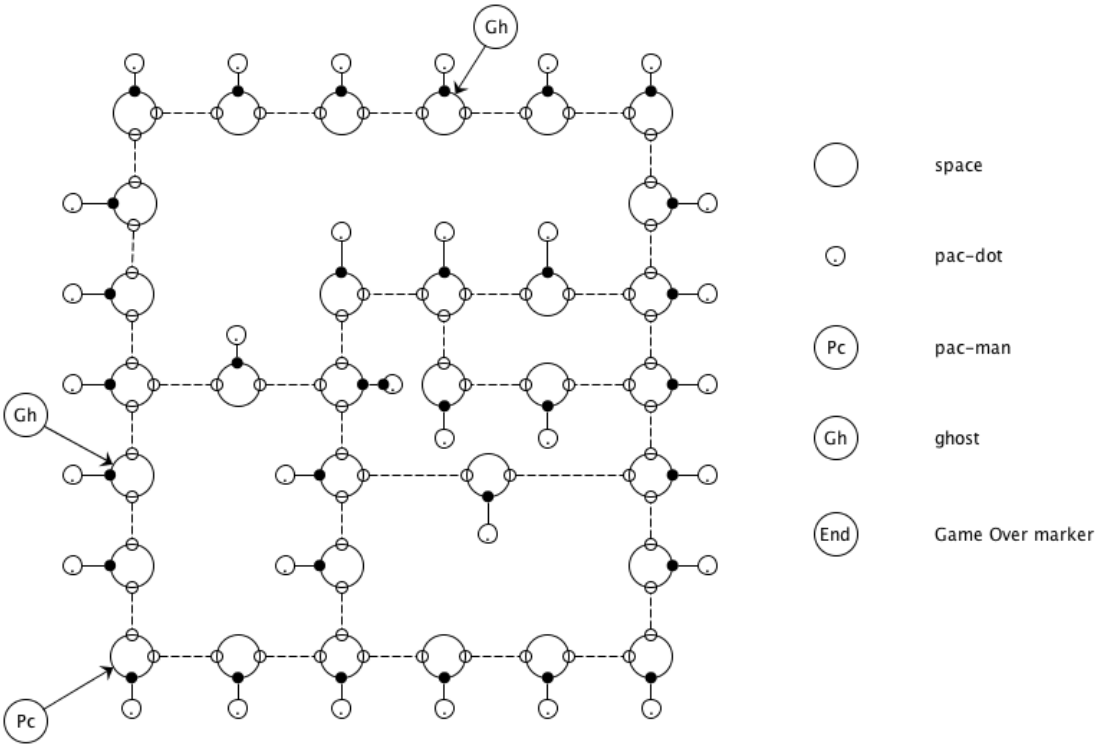


Figure 6.10: The pac-man playing field.

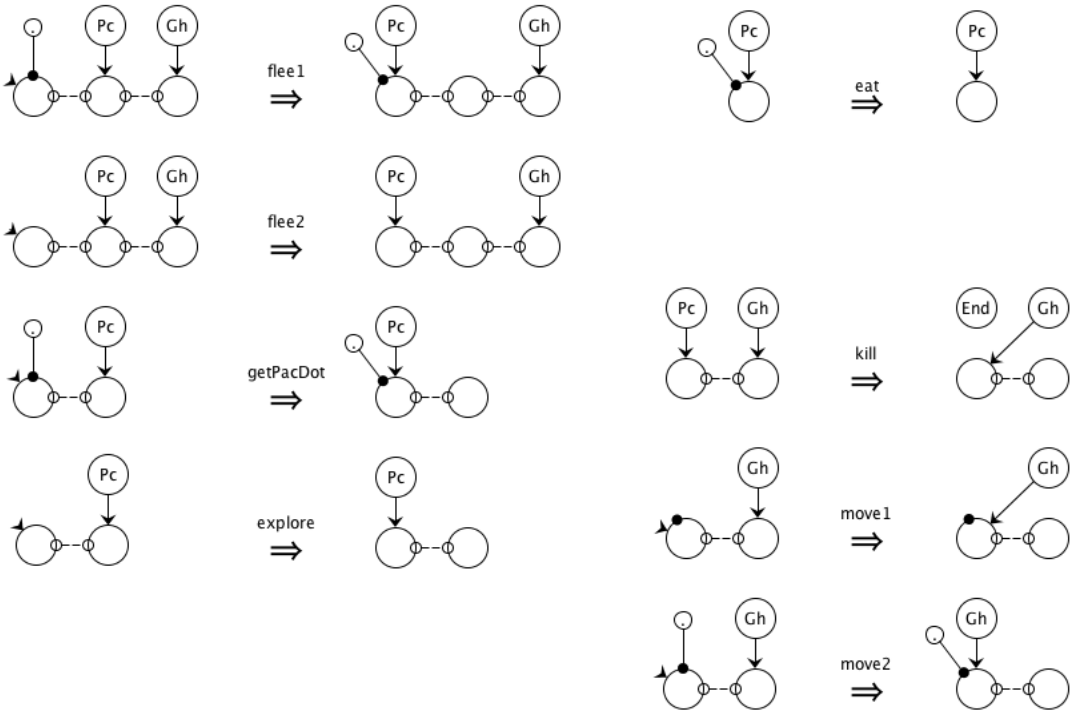


Figure 6.11: The set of rules to control pac-man (left) and to control the ghosts (right).

The strategy for controlling pac-man is as follows:

pacAI:

(flee1) `orelse` ((flee2) `orelse` ((getPacDot) `orelse` (`try`(explore))))

The rewrite rules for the ghosts can be found in Figure 6.11. Like for pac-mac, these rules and a set of strategies help simulate the AI to control the ghosts. A ghost's first priority is to eat pac-man (rule *kill*). If pac-man is not nearby, then a ghost tries to move to a space with no pac-dots (rule *move1*) since following empty spaces should lead the ghost to pac-man. If a ghost can only move to a space with a pac-dot, then do so (rule *move2*). The strategy for controlling ghosts is as follows:

ghostAI:

(kill) `orelse` (gMove)

gMove:

(move1) `orelse` (move2)

The overall strategy called *gameLoop* that controls the game is as follows: we must first check that pac-man has not been eaten (by checking for the existence of a node of type *End*). We then call *pacAI* followed by *ghostAI* for each ghost.

To make sure we only move each ghost only once per turn, we can do two of the following things:

- We can add all ghosts to P at the start of each game loop and make sure all the rules that involve ghosts have an empty M (i.e., no nodes from their right-hand sides will become part of the subgraph where the next rewrites will apply). This means every time a ghost performs an action, which removes the ghost from P, it cannot perform another one till the next game loop where we add all the ghosts back into P.

gameLoop1:

`repeat(`
`setPos(Property(`Pc`,CrtGraph)` $\cup$ `Property(`Gh`,CrtGraph)` $\cup$ `Property(`End`,CrtGraph));`
`if(isEmpty(Property(End,CrtGraph)))then(`pacAI ; `repeat(`ghostAI`))else(Fail)`
`)`

- Alternatively, we can not use the M subgraph of rules, and instead use the N subgraph by adding all ghosts on the right hand side of the rules into N. This means that once a ghost is moved, it is added to the *banned* subgraph and will not be able to be used for the rest of the turn. At the start of each turn, we then empty the *banned* subgraph to start the loop again.

gameLoop2:

`repeat(`
`setBan(`$\emptyset$`);`
`if(isEmpty(Property(End,CrtGraph)))then(`pacAI ; `repeat(`ghostAI`))else(Fail)`
`)`

In both gameLoop1 and gameLoop2, `repeat(ghostAI)` is guaranteed to be terminating since for the case of gameLoop1 the ghosts are removed from P after they are moved and so eventually no ghosts will remain in P. For gameLoop2, the ghosts are put into the *banned subgraph* after they are moved and so eventually no ghost rule will be applicable.

Running a graph program with this strategy would return a result set of all possible plays of that specific game of pac-man. If we only wanted one game to be returned we would only have to encompass each of the rules used in the strategy with a `one()` operator.

As we can see, a pac-man game with basic AI does not require many nodes and just six relatively simple strategies are sufficient to model it using our language. The position here is used like a *for loop* to iterate through all the ghosts once per turn. No extra rules (like rules that serve to increment counters) were needed to control the movement rules.

Adding a scoring system would be trivial: each time pac-man eats a pellet, a *point* node would be created and added to a *list* of points which can then be counted at the end of a game.

6.7 Pathfinding

We will now give a program to find a path in a labyrinth. The labyrinth is represented as a graph built out of *Labyrinth* nodes, as shown in Figure 6.12, where *Labyrinth* nodes are depicted as empty circles and exits are represented with an *End* node. The initial located graph in this example has a *Pather* node connected to the start of the maze.

A *Labyrinth* node has five ports, one for each cardinal direction North, East, South and West and a *Pather* port, where a *Pather* node can attach to (see Figure 6.13). The *End* node has the same ports as a *Labyrinth* node but will react differently when a *Pather* node is connected to it. We will also have a $Visited$ node, which has the same ports as a *Labyrinth* node but like the *End* node, will react differently when a *Pather* node connects to it. Lastly we have a $PATH$ node which has the same ports as the *Labyrinth* node and will be used to replace *Labyrinth* nodes so that a visible path will be drawn from the start to the exit of the labyrinth.

For the sake of clarity, in the following diagrams the four directional ports will not be labelled but will be drawn in the standard orientation on the nodes. In the rules, a white port means that the port must be connected, a crossed port must be free and a black port means either connected or free.

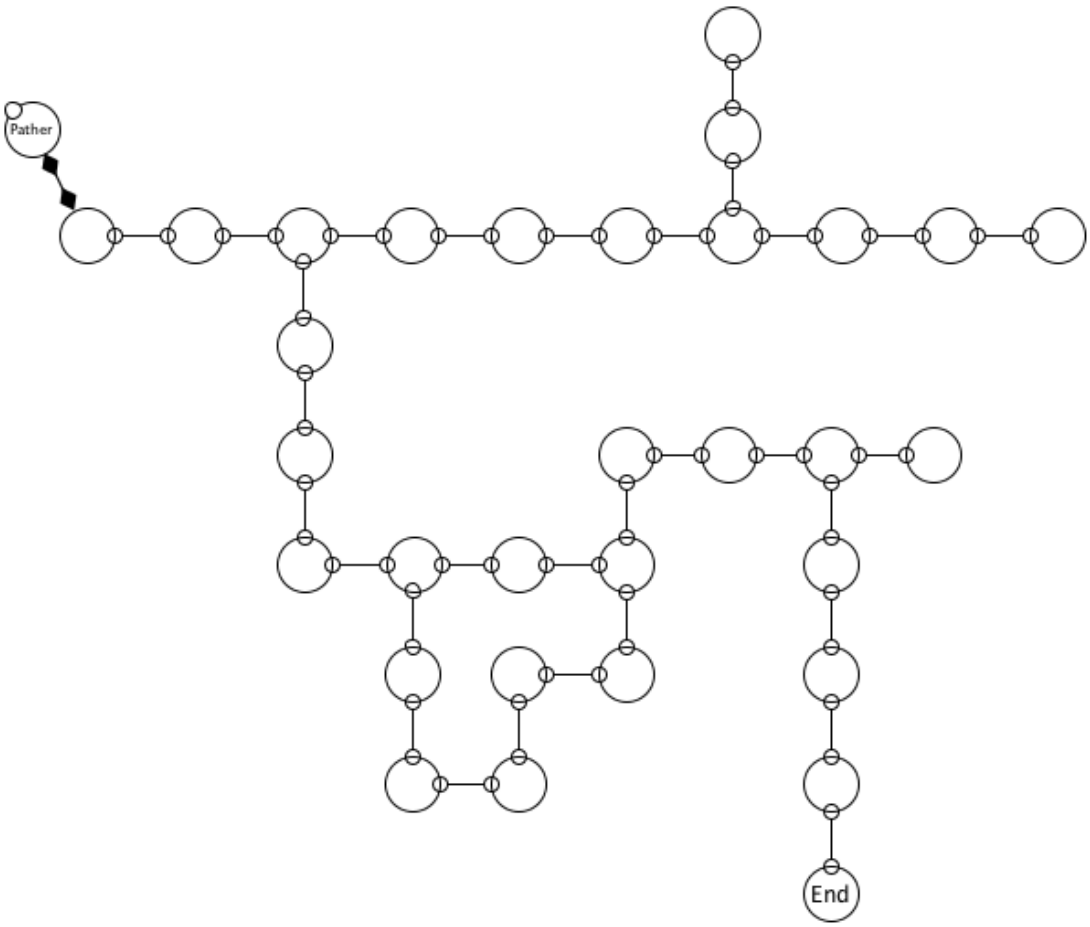


Figure 6.12: An example of a labyrinth.

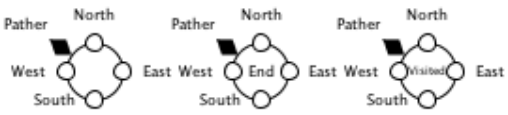


Figure 6.13: A *Labyrinth* node, *End* node and $Visited$ node.

A *Pather* node has a *Position* port and a *List* port. The *Position* port connects to a *Labyrinth* node and the *List* port will connect to a list of *Direction* nodes (representing the path followed so far). We have four *Direction* nodes N , E, S and W that each have two ports: a $Next$ and a $Prev$ port. We will also need a *Drawer* node (with the same ports as a *Pather* node) that will travel back to the start of the labyrinth, following a list of *Directions* and replacing *Labyrinth* nodes with $PATH$ nodes.

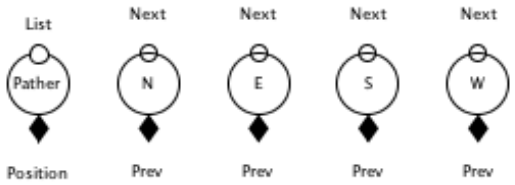


Figure 6.14: A *Pather* node and the four *Direction* nodes.

Lastly, we have some *management* nodes: two *copy* nodes (*cp*2 and *cp*3) and a *delete* node named ϵ. The *copy* nodes take a list of directions and duplicate (*cp*2) or triplicate (*cp*3) it. The ϵ node takes a list and deletes it. The rewrite rules for *cp*2 can be found in Figure 6.15 (the *cp*3 rules are similar but produce three copies) and the rewrite rules for ϵ in Figure 6.16.

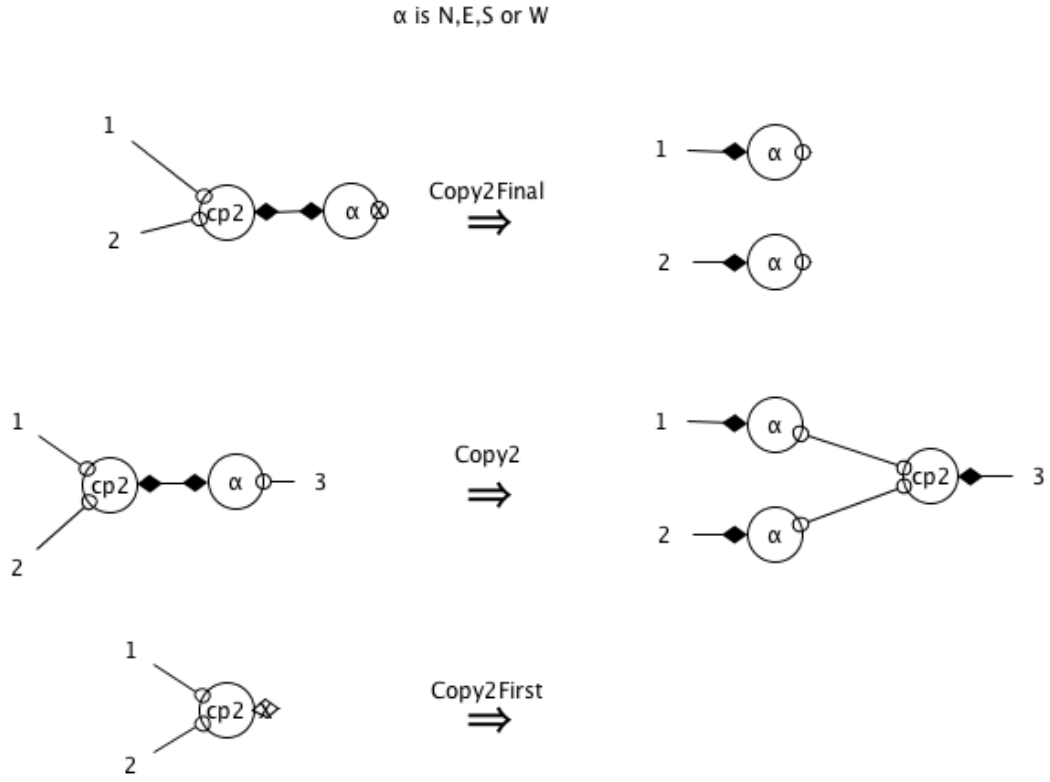


Figure 6.15: The set of rules for *cp*2.

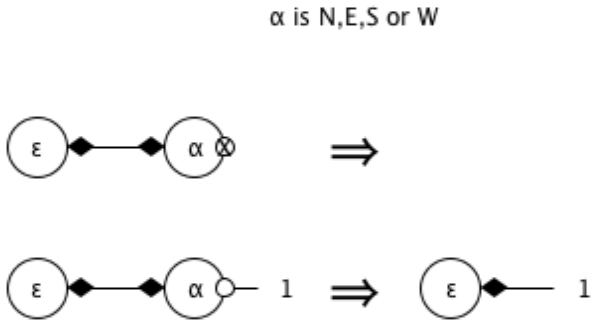


Figure 6.16: The set of rules for ϵ.

The program consists of the strategy expression *LabStrat* described below, and a located graph representing the labyrinth, where the only node in the initial subgraph P is the *Pather* node marking the starting point in the labyrinth. The strategy has two main parts, which we call Step 1 and Step 2. Step 1 attempts to find a path to the exit of the labyrinth, by moving the starting *Pather* until a *Pather* node positions itself onto the *End* node. If a *Pather* node is positioned onto an *End* node, a path was found and the program will move onto the Step 2. When a *Pather* node moves to a new position, it changes the *Labyrinth* node it moved from into a $Visited$ node. This will ensure the *Pather* node never backtracks.

The strategy will start by checking if a *Pather* node is connected to four *Labyrinth* or *End* nodes that have their *Pather* port free (rule *split*4 in Figure 6.17, a special case of when the starting *Pather* is put on such a *Labyrinth* node). This rule will remove the *Pather* node and create four new *Pather* nodes for each of the four positions and give each one of the new *Pathers* a corresponding *Direction* node (to remember the step done).

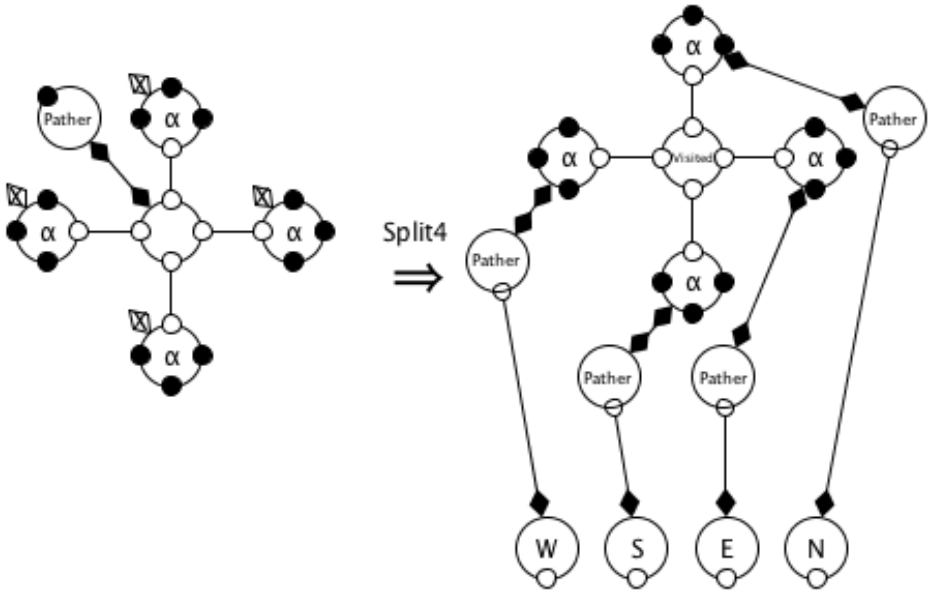


Figure 6.17: The *split*4 rule. α is a *Labyrinth* or *End* node.

If *split*4 cannot be applied, the strategy will try to apply one of the four *split*3 rules *split*3*a*, *split*3*b*, *split*3*c* or *split*3*d*. This rule deletes the original *Pather* node and creates three new *Pather* nodes, adding a corresponding *Direction* node to each of their lists, and copying the original *Pather* node's list onto the end of the new *Pather* nodes' lists. See Figure 6.18 for the *split*3*a* rule (the other three split3 rules are similar, taking into account the remaining combinations).

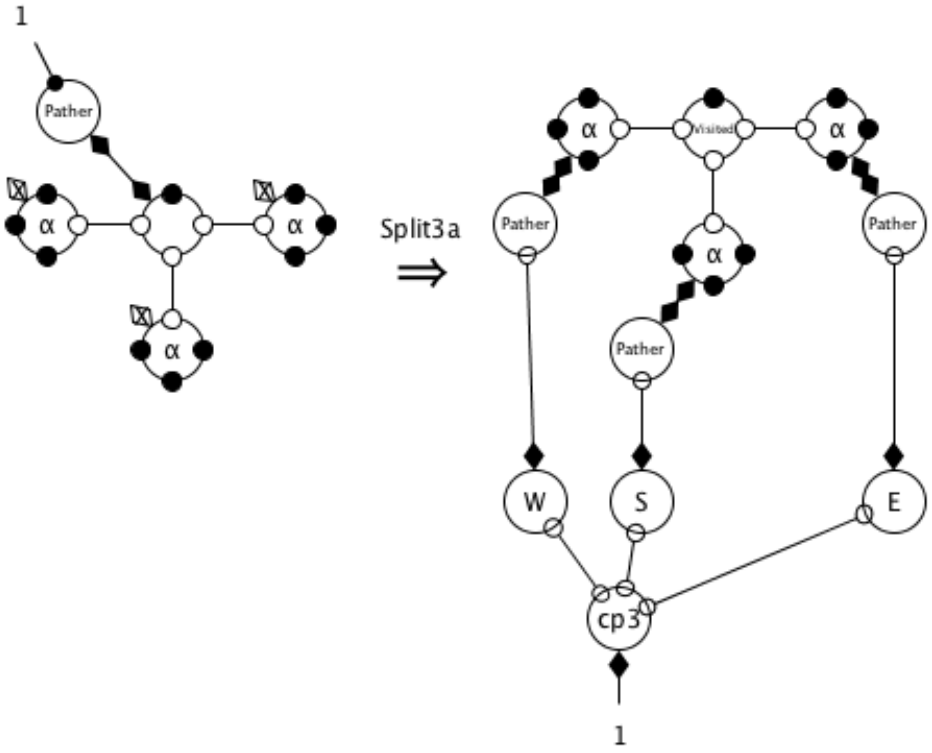


Figure 6.18: The *split*3*a* rule. α is a *Labyrinth* or *End* node.

If none of the *split*3 rules can be applied, the strategy then tries all six of the *split*2 rules and if none of the *split*2 rules can be applied, it tries one of the four *split*1 rules. These rules do the same thing as the *split*3 rules but only split to two and one *Labyrinth* nodes respectively. See Figure 6.19 for *split*1*a* and *split*2*a*.

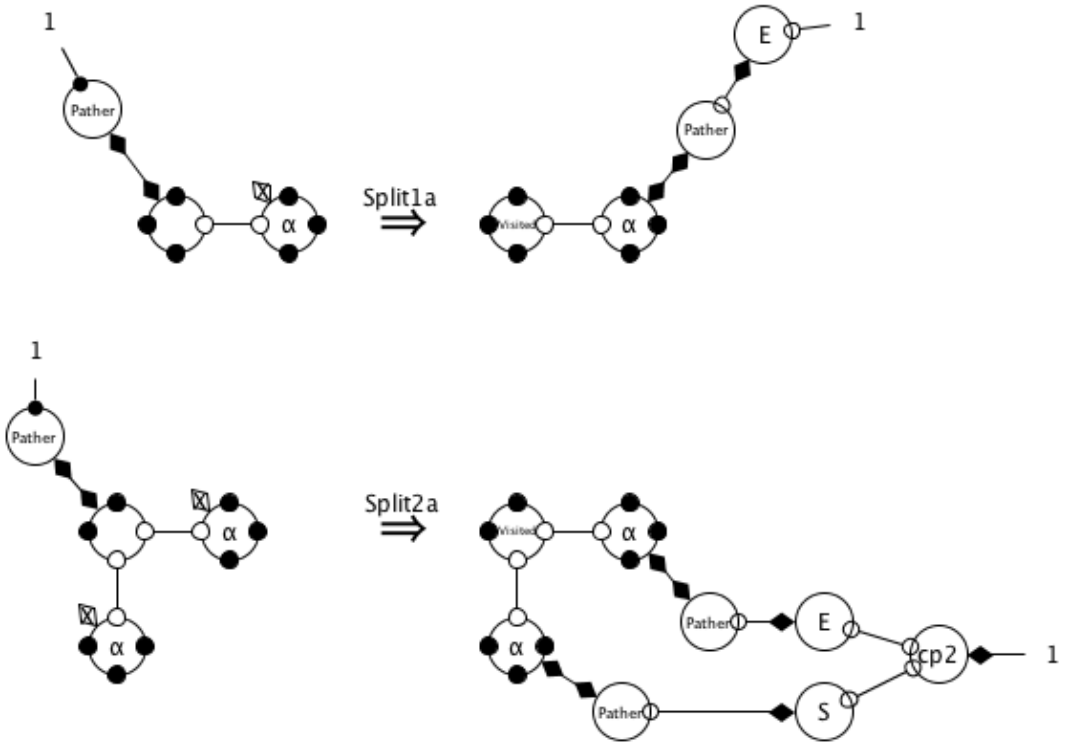


Figure 6.19: The *split*2*a* and *split*1*a* rule. α is a *Labyrinth* or *End* node.

We need to apply the *split* rules in this specific order or a possible split might be missed. For example, *split*1*a* might be applicable somewhere where *split*3*a* is also applicable but by applying *split*1*a* first we would not then explore the labyrinth to the West or South. This could lead to ending up with a longer path to the exit or in the worst case not finding the exit at all.

All *split* rules have an empty M subgraph. This will allow the strategy to move each *Pather* at most once per iteration. The strategy will do this and then use Property(ρ,G) to add all the *Pathers* back to P and then start over again. This ensures that no *Pather* is given priority and is needed to find the shortest path (as explained further down).

While trying to apply the *split* rules in that specific order, the strategy will constantly check if the $found$ rule (in Figure 6.20) is applicable. If it is, it moves onto Step 2: drawing the path. If the *End* node is not reachable from the starting point, the program will not terminate.

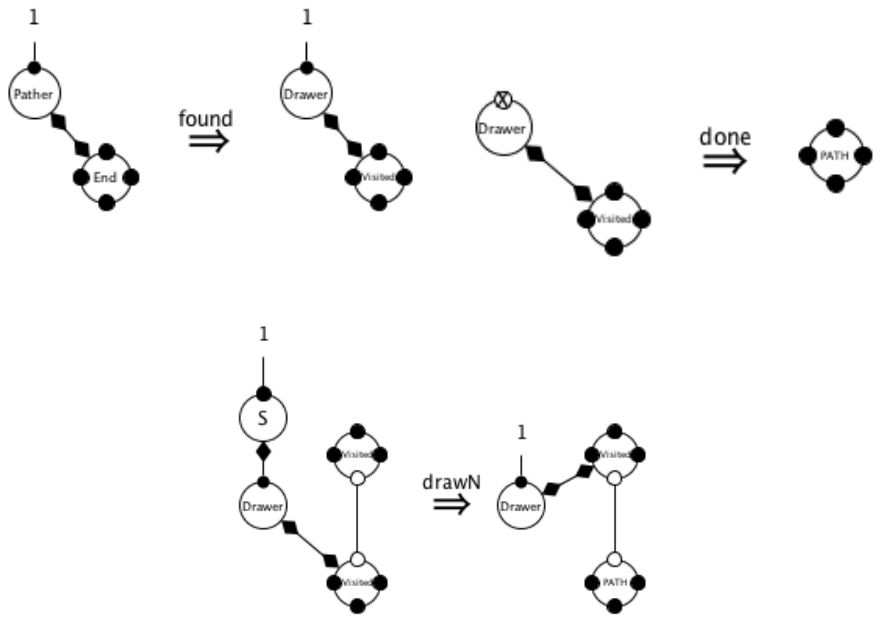


Figure 6.20: The *found*, *done* and *drawN* rules.

Step 2 checks if the *done* rule (Figure 6.20) is applicable and if it is not it will attempt to apply the four *draw* rules *drawN*, *drawE*, *drawS* and *drawW*. See Figure 6.20 for the *drawN* rule; the other three *draw* rules are similar and cater to a different direction each.

If the *done* rule is applicable, the program will terminate and our labyrinth will have the shortest path to the exit drawn on it.

LabStrat: Step1 ; Step2

Step1: `while(not(`*found*`))do(`
`setPos(Property(`Pather,`CrtGraph));`
`repeat(`Step1Split`);`
`) ;`
found

Step1Split: (*split*4) `orelse` ((*split*3*a*) `orelse` ((*split*3*b*) `orelse` ((*split*3*c*) `orelse` ((*split*3*d*) `orelse` ((*split*2*a*) `orelse` ((*split*2*b*) `orelse` ((*split*2*c*) `orelse` ((*split*2*d*) `orelse` ((*split*2*e*) `orelse` ((*split*2*f*) `orelse` ((*split*1*a*) `orelse` ((*split*1*b*) `orelse` ((*split*1*c*) `orelse` (*split*1*d*)))))))))))))

Step2: `while(not(`*done*`))do(`
(*drawN*) `orelse` ((*drawE*) `orelse` ((*drawS*) `orelse` (*drawW*)))
`) ;`
done

Since *Labyrinth* nodes are changed to *Visited* nodes when a *Pather* moves from them, if a branching occurs in the labyrinth that later reconnects (as seen at the lower middle Figure 6.12) the branch with the shortest path will be picked (remembering that each *Pather* can only take one

step at most during each iteration so the *Pather* in the shortest branch will get to the reconnecting *Labyrinth* node first).

Branching that reconnects will cause stuck *Pathers*. When the *Pather* from the quickest branch gets to the reconnecting *Labyrinth* node, it will split and go to the slowest branch. That newly split *Pather* will eventually meet the original *Pather* of that branch (going the other way). These two *Pather* nodes will be positioned in two adjacent *Labyrinth* nodes but won't be able to move and remain stuck there. We could extend our graph program by creating a set of rules to eliminate these stuck *Pather* nodes, using the ϵ node. This is mainly an aesthetic improvement since the stuck *Pather* nodes will not affect the functionality of the program.

This example uses the same concept as the pacman example (Section 6.6) where the *Pather* rules take the *Pather* nodes out of P so make sure that the `repeat(`Step1Split`)` always terminates. We can also use the same alternate concept as the packman example and use the *banned position* and make all the *Pather* rules put the *Pather* nodes into Q.

6.8 Flag Sorting

The following example shows how a non-ordered list of three colours (Blue, Red and White) can be sorted to represent the French flag (Blue first, then White and finally Red). We have three nodes representing each colour (shown in Figure 6.21) that have two ports each: a *previous* port (to the left of the node) and a *next* port (to the right of the node). We also have a *Mast* node at the beginning of the list.

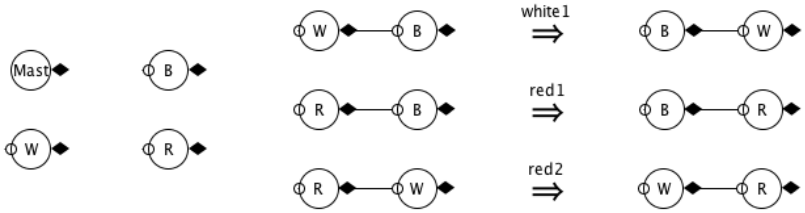


Figure 6.21: The four port node types and the 3 flag sorting rules.

Using the three rules in Figure 6.21, we can swap two colours if they are in the wrong order. Using the ||| operator we can also attempt to apply as many of these rules as we can in parallel. Our overall strategy would then be:

$$\texttt{repeat}(\texttt{setPos}(\texttt{CrtGraph}); ((\mathit{white1} ||| \mathit{red1}) ||| \mathit{red2}))$$

We can also program a sorting algorithm that starts from the mast. The initial P contains only the *Mast* node. If no rule can be applied, we move P one position across the list and try again. After a rule is applied we reset P to be just *Mast*. When we reach the end of the list, the program terminates and the list is correctly ordered. By defining: $\mathit{swap} \triangleq ((\mathit{white}1)\texttt{orelse}(\mathit{red}1))\texttt{orelse}(\mathit{red}2)$ and $\mathit{backToMast} \triangleq \texttt{Property}(\mathit{mast}, \texttt{CrtGraph})$, the strategy is:

`setPos(`*backToMast*`);`

```
while(not(isEmpty(CrtPos)))do
(if(swap)then(swap; setPos(backToMast))else(setPos(NextSuc(CrtPos))))
```

This example illustrates the modularity of programs in the language: to program a flag-sorting algorithm that proceeds from the mast onwards we do not need to change the rewrite rules (in contrast with other graph rewriting languages where focusing constructs are not available and conditional rewriting is used).

Appendix A shows figures of the Flag Sorting example running in PORGY.

6.9 Biochemical Reactions in the AKAP Model

An example can be found in [8] where a biochemical network of the scaffold-mediated crosstalk between the cyclic adenosine monophosphate (cAMP) and the Raf/MEK/ERK pathway [7] is modelled. This interaction has an important role in the regulation of cell proliferation, transformation and survival.

The six chemical species occurring in this model (AKAP, PKA, PDE8, Raf-1, cAMP and SA) are each represented by a node with ports representing sites on the species.

Four possible chemical reactions are written as rules in Figure 6.22:

- Rule $r1$ matches an AKAP connected to a PKA, a PDE8 and a Raf-1 (each with their right port free) and a cAMP that has its only port free as well. Two attribute conditions are also present in the left-hand-side: both right ports of PDE8 and Raf-1 must have an attribute *state* set to the value "−". The rule rewrites this left-hand-side to a same port graph where the cAMP port has been connected to the right port of the PKA.

- Rule $r2$ separates a cAMP from a PKA and changes the *state* attribute of the two right ports of PDE8 and Raf-1 to "+".

- Rule $r3$ matches a similar left-hand-side as $r1$ but in this case the *state* attribute of the right ports of PDE8 and Raf-1 must be "+". This rule then destroys the cAMP and creates a SA.

- Rule $r4$ looks for an unconnected cAMP and a PDE8 with a free right hand port (the connection status of its left port is unimportant) that has its *state* attribute set to "−". This rule destroys the cAMP molecule.

These four rules depend heavily on attributes, without them rules $r1$ and $r3$ would have the same left-hand-side. Creating these rules without attributes would require to have two nodes for PDE8 (PDE8+ and PDE8-) and two for Raf-1 (Raf-1+ and Raf-1-). Also if any new rule would not care about the state of PDE8's right port then two rules would need to be created to match either PDE8+ or PDE8-. These extra nodes and rules would expand to quite a large size if the nodes had more ports and more attributes since we'd have to cater to every possible combination of attributes. This shows the usefulness of attributes within nodes, ports and rules.

A simple strategy here would be to try to apply all the rules in parallel as many times as possible:

$$\texttt{repeat}(((r1 \,|||\, r2) \,|||\, r3) \,|||\, r4)$$

Figure 6.23 shows the starting model used for the application which contains a large number of cAMP nucleotides and some AKAP,PKA, PDE8 and Raf-1 species connected together. This screenshot is taken from the PORGY tool, described in the following chapter.

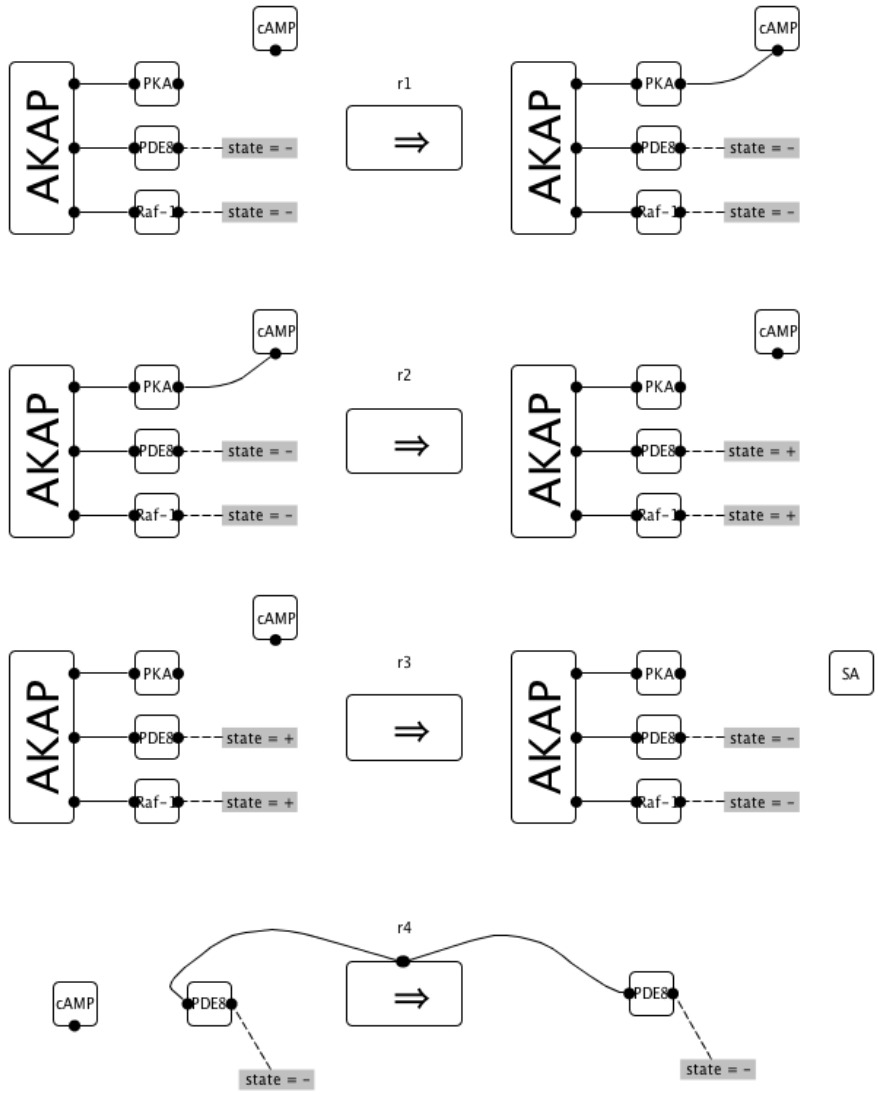


Figure 6.22: The rules relating to the AKAP Model.

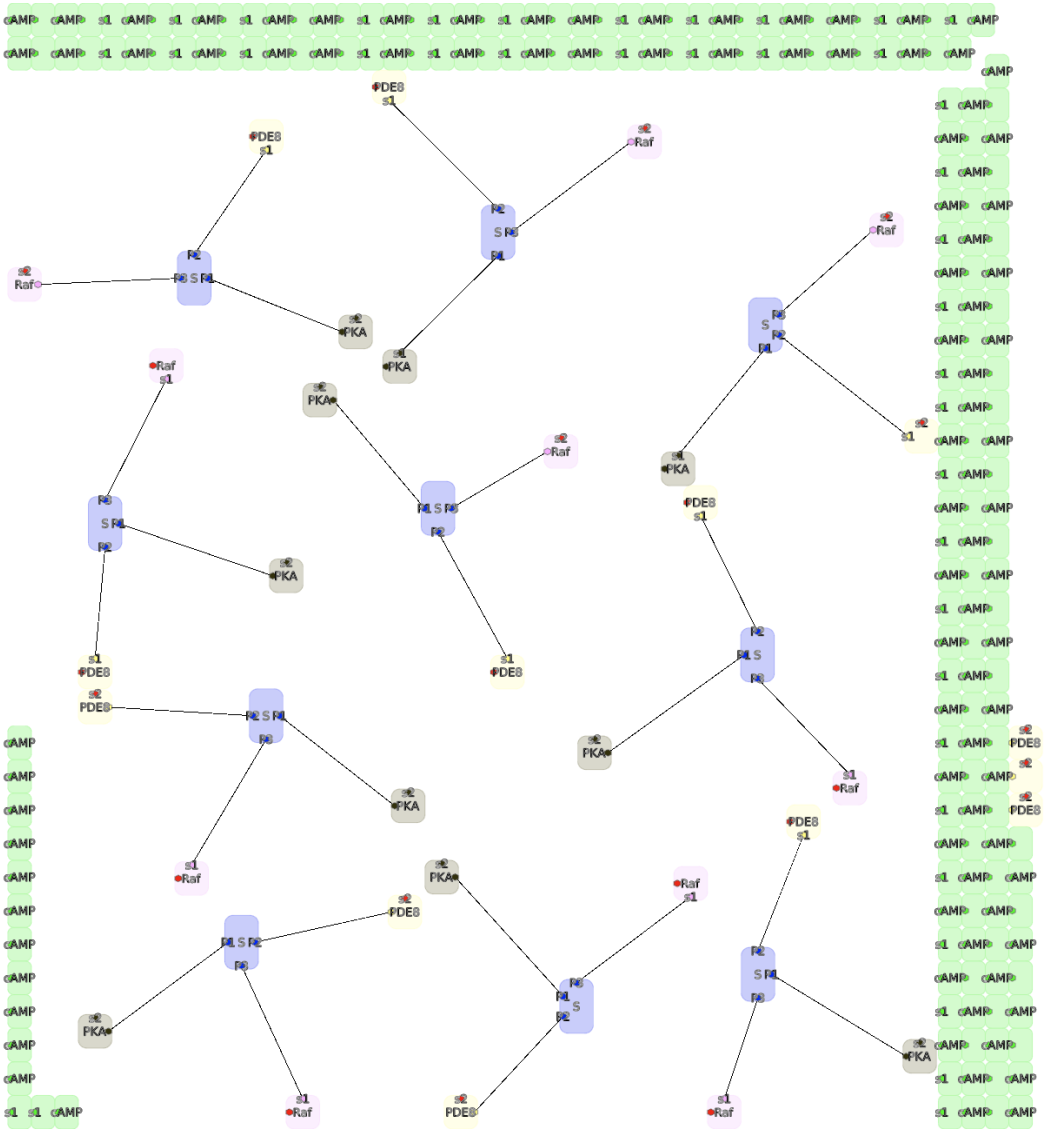


Figure 6.23: The starting graph for the AKAP model.

Chapter 7

PORGY

The PORGY Team

With previous research and interest in graph rewriting systems, the members of the PORGY team were interested in creating a tool to develop theoretical models of systems and create and execute them visually. Some of these models would require very large graphs and complex rules so a powerful tool was needed. LaBRI's TULIP[1] offered a strong backend for large models and a very powerful visual editor for graphs. The TULIP team, not having any features pertaining to graph rewriting, was interested in expanding its features to allow users to model and execute complex graph rewriting systems.

We will begin with an explanation of the capabilities of the TULIP tool and then continue with an explanation of what the PORGY tool is and its relation to TULIP. We then present a feature set for PORGY and explanations of those features.

The author's main contributions for PORGY from an implementation point of view, is the Strategy Engine and making design decisions on the GUI and interface of PORGY.

7.1 The TULIP Tool

> Tulip is an information visualisation framework dedicated to the analysis and visualisation of relational data. Tulip aims to provide the developer with a complete library, supporting the design of interactive information visualisation applications for relational data that can be tailored to the problems he or she is addressing.[2]

Tulip, created by David Auber[2][13], is a multi-platform tool entirely coded in C++. It uses a powerful backend to store and manage potentially very large graphs and OpenGL[3][70] to display these graphs with speed and visual flair.

TULIP is plugin based, and many built-in plugins allow users to analyse data from graphs and to run display algorithms on graphs to change the way they are drawn and structured. This highly modular plugin based system would make it very easy to create and implement the necessary graph

[1] http://tulip.labri.fr/TulipDrupal/
[2] http://www.labri.fr/perso/auber/
[3] http://www.opengl.org/

rewriting algorithms needed into TULIP to create the PORGY tool. The interface is also highly modular and can easily be modified to prioritise certain interactions such as applying rules and creating strategies.

TULIP also allows users to create and assign properties to nodes and edges and to create subgraphs within graphs. There is however no notion of ports in the TULIP graph system and so a layer had to be implemented between the frontend and the backend to model port graphs. Other notions such as derivation trees are also not present and had to be implemented.

7.2 The PORGY Tool

One of the goals of PORGY is to allow the user to interact and experiment with a port graph rewriting system in a visual and interactive way. The user should be able to interact with the port graph rewriting system at any of these stages:

- Creation of node types.
- Rule creation.
- Creating initial graph.
- Coding different strategies.
- Running strategies and applying individual rules to populate the derivation tree.
- Debug and analyse the resulting derivation tree.

Ideally, interaction should be as visual as possible, for example dragging a rule from a *rule catalogue* onto a graph to try apply the rule to that graph.

In the following subsections, *graphs* (such as the initial graph and all subsequent derived graphs) are referred to as *models* since the word *graph* is used in the backend of TULIP to define graphs and subgraphs as data structures; we use the word *model* internally to avoid confusion.

Before explaining the different aspects of the tool, we need to define how we will implement ports (and therefore port graphs), rules, models and derivation trees (also called trace trees) into TULIP. Figures for the following sections can be found in Appendix B.

7.2.1 Implementing Ports, Rules and Models

In TULIP, a graph is defined as a set of nodes and a set of edges. Edges are represented by a pair of nodes (Figure 7.1). There is therefore no built-in notion of ports.

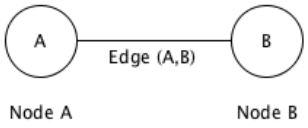


Figure 7.1: TULIP's representation of graphs.

To simulate a port node with n ports, we create a main TULIP node for the port node itself, and then n other TULIP nodes as the n ports that we connect to the main TULIP node with edges (Figure 7.2).

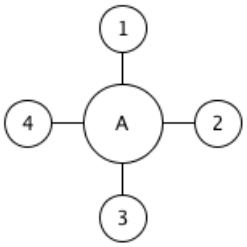


Figure 7.2: A port node A (with its four ports 1,2,3 and 4) represented in TULIP.

Methods were then coded so that plugins can go from port node form to TULIP form. Users can then create and edit nodes, rules and graphs as port graphs and the plugins can then convert back into TULIP form for the backend. A visualisation algorithm is also created that takes a port node in TULIP form and displays the ports internally to the node (Figure 7.3, in this specific example the ports are of a darker colour and some are of a different shape). This means that the user never explicitly sees or interacts with elements that are not in a port graph form.

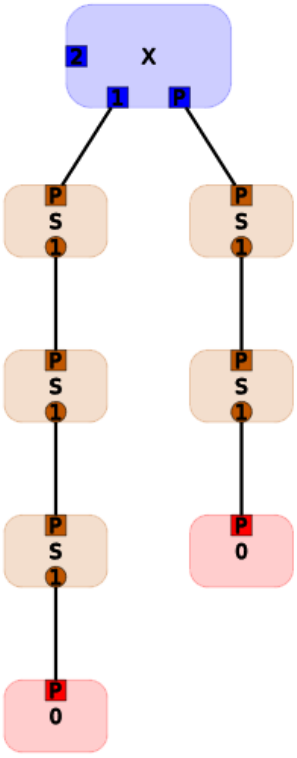


Figure 7.3: An interaction net example (3×2) displayed using the port graph drawing algorithm.

TULIP has a powerful construct of subgraphs that can be created from other graphs or subgraphs. The entire PORGY hierarchy uses this notion of subgraphs to organise and store the rules, models and derivation trees. The starting universal graph is called the *root* and has three distinct subgraphs for rules, traces and models. Each of these subgraphs themselves are composed

of subgraphs used essentially as a list of their elements. For example, the rules R_1, R_2 and R_3 will be subgraphs of the *Rules* graph (itself a subgraph of the *root* graph). For more details on this, please refer to the technical documentation found in [1].

Using this system of subgraphs, our plugins can access lists of the different element types used in PORGY while still maintaining the highly efficient (in terms of speed and memory space) data structure built into TULIP.

Rules are represented as a subgraph that contains an *arrow node*. All nodes added to a rule have a property *Rule_side* that must be set to either $left$ or $right$. The *arrow node* will have a port for each interface link between a left hand side node and a right hand side node. This is then used in the rewriting plugin for the rewiring.

7.2.2 Node, Rule and Graph Creation

graphPaper is not the only way users can create their node types, rules and graphs. PORGY comes with a built in editor that is less visually intuitive but more useful if you then need to generate an initial graphs that contains a very large number of identical initial nodes (for instance, the biochemical in Section 6.9 has a large number of CAMP nodes to create.)

The main creation window (Figure B.1) is divided in three main parts: *node* creation, *model* creation and *rule* creation.

Nodes: A popup window appears for each created node and its ports (Figures B.2 and B.3). Properties such as name and colour can be defined for the node and ports can be added to its list of ports. Like for nodes, users can set name and colour properties for each port and can also choose a glyph (the shape that will be drawn to screen, for example a circle, square, hexagon...).

Models: Created nodes can be added to the initial model. A pop up window allows the user to select a node and how many instances of it to add to the initial graph Figure B.4). It is not possible at this stage to connect nodes together, this can be done after the creation process manually by creating edges between ports on in the visualisation of the model. This is very useful for certain applications such a biomolecular examples where large quantities of unconnected nodes need to be in the starting model (Section 6.9).

Rules: Rule creation happens in two stages. A first window allows the user to give the rule a name and has two lists of nodes, one for each side of the rule (Figure B.5) that the user can add nodes to. Once all the nodes are added, the user can click on the *Edges* button to create edges between nodes of each side and then edges between the rule nodes and the *arrow node* to create the interface (Figure B.6 and B.7).

These windows were all created for PORGY as TULIP plugins. For the manual editing of models, users can bring up the view of any models and use the *add node* and *draw edge* buttons to edit a model. *add node* lets a user select a node type from a list (and even create new types of nodes) and then click anywhere in the model to add it. *draw edge* allows the user to the click on a first port and then a second port to create an edge between them. A *delete* button is also available

to remove any nodes and edges (*dangling* edges resulting from deleting a node are automatically deleted.

Adding nodes to a rule happens in the same manner except that the user must choose if the nodes he is about to add should go into the right hand side or left hand side of the rule.To add edges that define the interface between both sides of a rule, the user simply uses the *draw edge* button and PORGY will automatically create a port on the *arrow node* and draw the proper edges.

7.2.3 The Main PORGY Window

The main PORGY view is displayed as a perspective which is similar to a plugin but used to create custom interfaces for TULIP. It is divided into a main viewing window and then a modular collection of windows that display information such as a list of rules, an editor and launcher for strategies...(Figure B.8).

By default Trace view is displayed, this is where the user will be able to see the derivation tree that is created and updated after each rule application or strategy execution. A black arrow from one model to another represents a single rule application or a parallel application of rules while a green line shows the application of a strategy to make it easier for the user to visualise the end point of a strategy (especially useful if there were many rule application in the strategy in question). If a strategy or application is successful the arrow will point to the final model created from the application of the rule or strategy. On the other hand, a fail application of a rule or strategy will point to a red square, a visually explicit representation of failure. An example of a trace can be seen in Figure B.9.

Using TULIP features, the trace is essentially a *graph* using special nodes called meta-nodes. These meta-nodes can be linked to our models and are visually a *window* into the model: zooming into the meta-node the user will be able to see the model it is representing and the user can even *enter* the window to gain access to the model.

Models, rule and a rule catalogue can be loaded up into their own windows so that users can visualise and interact with them. The rule catalogue is especially useful for rule application, as explained in the following section.

7.2.4 Model and Rule Visualisation

TULIP come with different display algorithm for different types of graphs and PORGY uses some of those display algorithms to draw models. Depending on the application, a particular algorithm can be used to draw the model, for instance the Flag Sorting example in Section 6.8 uses the Sugiyama (OGDF) algorithm to draw the list of colours in a straight line while the GEM algorithm gives better visual results for the biomolecular example in Section 6.9. Users can create their own display algorithms to suit the visual style of what they are modelling.

Rules have a special drawing algorithm that forces nodes to always be drawn on the correct side of the arrow node and to be spaced in a way to preserve overall shape between left and right hand side as much as possible.

7.2.5 Strategy and Rule Application

Rules can be applied through the *Algorithm* → *General* → *Check and Apply Rule* menu item (Figure B.10) where the user can enter the name of the rule they want to apply or by directly dragging the rule from the *rule list* or *rule catalog* onto a model. This more intuitive method still opens up the same window seen in Figure B.10 but fills in the window according to which rule was dragged. This method of interaction gives the user a feeling of direct feedback which helps with visualising and interpreting the changes that happen in the trace tree after a rule application.

Strategies are created and edited in the *Strategies* pane (Figure B.11). The user selects a strategy from a list and can then edit the strategy directly in a text box. They can then manually select a starting model and click the *Execute* button to apply the strategy to that model or (like for rule application) directly drag a strategy from the strategy list onto a model. It is also possible to generate strategies from the trace tree: the user selects a sequence of models on the tree and then clicks on the *emph* icon in the *Strategies* pane and then chooses *Strategy from trace graph*. This will create a strategy that will be a sequence of rules that derive the starting model of the sequenced to the end model of the sequence.

Strategies can also be imported from a *.txt* file and exported to one as well. PORGY has added macros into the strategy language so that a strategy *strat1* can be written into another strategy by typing #*strat1*# into it.

After a rule or strategy application, the view of the trace tree changes by zooming in/out and panning to show the the single derivation set (in the case of a rule application) or the whole branch created by a strategy. This, much like the rule and strategy dragging, helps the user see where and how an application changed the trace tree and is very helpful in debugging and analysing the modelled system.

7.2.6 Debugging and Static Analysis

PORGY comes with a set of features to debug and analyse a modelled system. A branch or branch section can be selected and then visualised in different ways: a *small multiples view* displays the sequence as a detailed step by step (including highlighting the left hand side instance, the left hand side removal and the right hand side insertion) grid showing the evolution of the sequence (Figure B.12). The user can then also watch and control an animation of the evolution of the sequence which can help them better see the effects of the derivations (Figure B.13). A branch or branch section can also be viewed as a localised sub trace (so that the user can visually focus on it) or a histogram where the evolution of certain properties (such as the evolution of the quantity of different types of nodes) can be analysed.

Under a special *Get Information* mode, highlighting models in a trace will display the strategy needed to get from the initial model to the highlighted one. Highlighting black arrows between models will tell the user what rule was applied, which left hand side instance was used and which *position subgraph* was used to generate the new model. This is especially useful to differentiate models derived from the same original model, but that used a different rule, left hand side instance or *position subgraph*. Highlighting a green arrow will show the user what strategy was applied to

Algorithm 1 Algorithm for matching the left-hand side R_L of a rule R in a model G

```
R_L ← Left-hand side of the rule to apply
n ← number of instances to return
for all connected component c of R_L do
  if c is a single port node then
    Find all equivalent instances of this port node in G
    if no instance found then
      return fail
    end if
  else if c is composed of an edge linking two port nodes then
    Find all equivalent instances of this pattern in G
    if no instance found then
      return fail
    end if
  else if c is composed of at least two edges then
    Find all equivalent instances of each edge of c in G
    Only keep the solutions with only one connected component and edges in the same order
    if no instance found then
      return fail
    end if
  end if
  Construct all instances of R_L in G by using the instances of each connected component of R_L
  in G
  return n (or all) constructed instances
end for
```

the starting model to produce the end model and what the starting *position subgraph* was.

User can also select subgraphs within a model thus highlighting them not only in that model but also highlighting all the elements of that subgraph in all subsequent models that contain them. This is particularly useful to follow the *life-span* or a particular node, edge or even subgraph.

Another feature merges all models that are isomorphic: this visually shows cycles between models and if there are multiple paths between two models, which one is the shortest.

7.2.7 Pattern Matching

The pattern matching plugin was built to deal with the particularities of graph rewriting within *located graphs* and is mainly inspired by Ullmann's original algorithm [74]. The plugin must not only find all the instances of the left hand side of a rule but also make sure that the instances overlap with the *position subgraph* and have nothing in common with the *banned subgraph* (Both of these subgraphs are defined in Section 5.1.1).

Algorithm 1 from the technical documentation in [1] matches the left hand side R_L of a rule R in a model G:

When an instance L_i is found, the plugin then checks that $L_i \cap P \neq \emptyset$ and $L_i \cap Q = \emptyset$ and then creates a subgraph for the rule instance on the model subgraph. The subgraph is labelled with the name of the rule being applied and which P and Q were used. This creates a list of possible applications and is useful to avoid reusing the pattern matching algorithm if the user decides to later apply the rule again on a same model (with a same P and Q).

For now, this brute force and unoptimised pattern matching has been implemented because having a working pattern matching algorithm was essential to all the other aspects of PORGY (rule application, the strategy engine...). Further optimising will be done such as implementing the check against the *banned subgraph* during the algorithm and not after a pattern has been matched and only starting the search from within the *position* subgraph. A specific matching algorithm could be created for Interaction Nets since the matching of the left-hand-side of a rule would only have to find two nodes connected by their principal port, making the matching considerably easier. Other pattern matching algorithms and methods could also be looked into to see if their compatibility with the TULIP back-end could bring further optimisations such as Cordella[28], Lipets[53] and Olmos[63].

Matching of attributes and values is not currently implemented. Adding that into the algorithm would mean that *finding all equivalent instances of this node* would mean not only matching the label but also matching the set of attribute-value pairs.

Other tools such as PROGRES, Fujaba and AGG (detailed in Chapter 3.2) have different pattern matching algorithms dependant of their data structure and the specific features of their rules and rewriting steps. A comparison of those can be found in [43].

7.2.8 Rewriting Plugin

The rewriting plugin takes a subgraph containing a instance of the left hand side of a rule and uses the rule from its name to perform a rewriting step.

It creates a new model which is an exact copy of the current model and then checks the interface of the instance of the left hand side and stores it temporarily. The left hand side instance is then deleted and an instance of the right hand side is added to the new model. The plugin then checks the saved interface and reconnects it to the interface of the right hand side instance. It finally adds a node between both models containing the name of the rule and the specific instance of the left hand side that it used for the rewrite step.

7.2.9 Strategy Engine

The strategy engine is a plugin that takes as an input a string containing a strategy and a port graph (by selecting a PORGY model within the UI). A list of user defined *position* and *banned* subgraphs then appears and the use can select a *position* and a *banned* subgraph. It then creates a vector containing the strategy (much like an abstract syntax tree). It then checks that the syntax is valid and any variables (such as rule names and property names) exist. The parser part of the strategy engine uses the Boost libraries[4] which contains classes and libraries that help parsing

[4]http://www.boost.org/

languages, giving us white space and new line handling for free.

A *run* function is then executed on the the top element of the vector which tries to apply a hardcoded version of the semantic rules from Section 5.3. Every time the strategy engine gets to a rule application, the Pattern Matching plugin (described in Section 7.2.7) is called and if a match is found, the rewriting plugin (described in Section 7.2.8) is called on one of the instances of a match (non-deterministically). If the rule isn't applicable then a *fail* model is created.

When the strategy engine terminates, it connects the starting model to the end model by a green arrow which contains a string of the strategy applied.

Currently, PORGY applies all rules as if they were in a `one()` construct therefore producing a single branch each time a rule or strategy is executed. Branching on all possible applications of a rule is currently in development to ensure soundness and completeness in regards to the semantics defined in Chapter 5 (for example, the `if()then()else()` in its current form is not sound. Since it does not have back-tracking, if the execution of the condition fails then the then-branch is executed. An exhaustive execution of the condition would solve this.). When implemented, the leaves of the derivation tree will be all the values of the *result set* of the graph program (if the program is strongly terminating). Currently, if users execute a same strategy multiple times on a starting model, the set of leaves of the resulting trace tree is a visual representation of a subset of the *result set* defined in Chapter 5.

Chapter 8

Conclusion & Future Work

In this thesis, we introduced a strategy language for graph rewriting and two tools graphPaper and PORGY to create graph rewriting systems and to execute strategies onto them.

graphPaper and PORGY are both tools that have been designed with visualisation and ease of use in mind.

This thesis makes several contributions:

- Located graphs are introduced as port graphs that contain two subgraphs that affect rewriting: the *position subgraph* P must have an intersection with the instance of the left hand side of the rule being applied while the *banned subgraph* Q must have nothing in common with that left hand side instance. Combined with the focusing constructs of the strategy language, it provides expressive power (selective rule application, complex graph traversals) while still remaining intuitive to use. Dividing the language into four syntactic categories (see Section 5.2) that are distinct except for well defined bridges (`setPos()`, `setBan()`, and different forms of rule application in strategies) emphasises the specific use of each operator and makes strategies easier to read and write. The notion of a *result set* is introduced and fits in nicely with the practical use of trace trees (intuitively, the result set corresponds to all the leaves of a trace tree).

- graphPaper provides a clean and focused environment to users to easily create port graph and rules with a simple (yet powerful) context based interaction method. graphPaper aims to have user interaction as similar as creating port graphs with a pen and paper, with the added benefit of dynamic digital information.

- PORGY, with the powerful TULIP backend, allows users to apply rules and strategies to port graphs of potentially large size and see the results in a trace tree. All of PORGY's information is displayed in a graphical way which fits nicely with the visual nature of graphs. Many debugging and data analysis features exist such as animating a sequence of rewrites and generating strategies from the branch of a trace tree. All these features combined give users a powerful and versatile environment with which to model strategic graph rewriting systems.

Strategy Language

Graphs and graph rewriting has shown itself as a powerful formalism to model various different kinds of systems ranging from biomolecular systems to an AI driven game of pacman. These models also need a control method that governs the rewriting: the strategy language described in this thesis is strongly inspired by the work on GP and PROGRES, and by the strategy languages developed for term rewriting. It can be applied to terms as a particular case (since terms are just trees). When applied to trees, the constructs dealing with applications and strategies are similar to those found in ELAN or Stratego, where users can also define strategies to apply rules in sequence, to iterate rules, etc. The focusing sublanguage on the other hand can be seen as a lower level version of these languages, because term traversals are not directly available in our language but can be programmed using focusing constructs. By clearly separating the Focusing, Application and Strategic constructs of our strategy language (except for well defined bridges such as `setPos()`), users construct the rules, traversals and control method in a clear manner and simpler manner with few overlaps. The notion of results sets, due to the fact that a given model can evolve in different ways, can be used well in implementations as part of derivation trees.

At the moment, the strategy language is a minimum kernel for strategic graph programming, but in the future will be extended with high-level programming language features such as types, modules, libraries, focusing variables... A ppick() operator could be added that would take a set of strategies and associated probabilities and and pick one of them according to their probability (this would require a definition of a probabilistic transition relation in the semantics).

Rules could be extended with conditions and the ability to transfer attribute values from the left to the right hand side of a rewrite (the latter could be done by removing the $*$ element and creating $\mathcal{X}_{\mathscr{V}}$, a set of variable value labels, and an additional phase within the rewrite step would also be necessary to transfer the values from the left-hand-side to the right-hand-side).

Values could be extended to support expressions that in turn would need to be evaluated. This would allow attributes to be, for example, equal to "4+1" and could even be extended to allow code to be executed that would then return a value to the attribute.

graphPaper & PORGY

The graphPaper tool is still under development with rule creation and exporting to PORGY being actively worked on. PORGY, using the strategy language in this thesis, has a strong and developed control mechanism as well as a powerful and intuitive visualisation (up to 5,000,000 nodes at once) and interaction aspect. Various verification, analysis and debugging tools allow users to refine their models, rules and strategies after an execution all within the same tool, proving to be highly efficient.

Certain parts of the strategy language such as the `Property(,)` construct are only partially implemented and currently backtracking is still in development (meaning that all rules are applied using `one()` for the moment). The addition of a more feature rich strategy editor is planed (with syntax highlighting) and the ability to add break-points in a strategy to help with debugging. Work is also being done on the interface to allow users to create *position* and *banned* subgraphs

on models and to choose which of these subgraphs to use when applying a single rule or a strategy. A more general matching algorithm is being looked into to allow macros in rules and higher order variables.

From the visualisation point of view, we are working on enhancing the algorithms for drawing rules and models, and we already have begun to work on the mental map preservation. Moreover we will address other application domains: for instance linguistics analysis [42] shares some of the features of biological networks and we expect to be able to handle linguistic models in PORGY without much difficulty. Specific perspectives could also be created for specific model types to better suit the visualisation and interaction on these models.

This concludes the thesis.

Bibliography

[1] https://gforge.inria.fr/projects/porgy/. *PORGY website*.

[2] http://tulip.labri.fr/tulipdrupal/. *TULIP Website*.

[3] http://www.oliviernamet.co.uk/. *graphPaper Details*.

[4] J. Abello, F. van Ham, and N. Krishnan. Ask-graphview: A large scale graph visualization system. *IEEE Trans. Vis. Comput. Graph.*, 12(5):669–676, 2006.

[5] P. Andersson. Introduction to hyperedge replacement grammars.

[6] O. Andrei. *A Rewriting Calculus for Graphs: Applications to Biology and Autonomous Systems*. PhD thesis, Institut National Polytechnique de Lorraine, 2008.

[7] O. Andrei and M. Calder. A model and analysis of the akap scaffold. *Electr. Notes Theor. Comput. Sci.*, 268:3–15, 2010.

[8] O. Andrei, M. Fernández, H. Kirchner, G. Melançon, O. Namet, and B. Pinaud. PORGY: Strategy-driven interactive transformation of graphs. In R. Echahed, editor, *TERMGRAPH*, volume 48 of *EPTCS*, pages 54–68, 2011.

[9] O. Andrei and H. Kirchner. Graph rewriting and strategies for modeling biochemical networks. In V. Negru, T. Jebelean, D. Petcu, and D. Zaharie, editors, *SYNASC*, pages 407–414. IEEE Computer Society, 2007.

[10] O. Andrei and H. Kirchner. A Rewriting Calculus for Multigraphs with Ports. In *Proceedings of RULE'07*, volume 219 of *Electronic Notes in Theoretical Computer Science*, pages 67–82, 2008.

[11] O. Andrei and H. Kirchner. A Higher-Order Graph Calculus for Autonomic Computing. In *Graph Theory, Computational Intelligence and Thought. Golumbic Festschrift*, volume 5420 of *Lecture Notes in Computer Science*, pages 15–26. Springer, 2009.

[12] D. Auber. Tulip. In *Graph Drawing*, pages 435–437, 2001.

[13] D. Auber. Tulip : A huge graph visualisation framework. In P. Mutzel and M. Jünger, editors, *Graph Drawing Softwares*, Mathematics and Visualization, pages 105–126. Springer-Verlag, 2003.

[14] D. Auber, P. Mary, M. Mathiaut, J. Dubois, A. Lambert, D. Archambault, R. Bourqui, B. Pinaud, M. Delest, and G. Melançon. Tulip : a scalable graph visualization framework. In S. B. Yahia and J.-M. Petit, editors, *EGC*, volume RNTI-E-19 of *Revue des Nouvelles Technologies de l'Information*, pages 623–624. Cépaduès-Éditions, 2010.

[15] F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge University Press, Great Britain, 1998.

[16] D. Balasubramanian, A. Narayanan, C. P. van Buskirk, and G. Karsai. The Graph Rewriting and Transformation Language: GReAT. *ECEASST*, 1, 2006.

[17] E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles. Tom: Piggybacking Rewriting on Java. In F. Baader, editor, *RTA*, volume 4533 of *LNCS*, pages 36–47. Springer, 2007.

[18] H. Barendregt. *The Lambda Calculus, Its Syntax and Semantics.* North Holland, 1981.

[19] H. P. Barendregt, M. C. J. D. van Eekelen, J. R. W. Glauert, R. Kennaway, M. J. Plasmeijer, and M. R. Sleep. Term graph rewriting. In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, *PARLE (2)*, volume 259 of *Lecture Notes in Computer Science*, pages 141–158. Springer, 1987.

[20] S. Barker and M. Fernández. Term rewriting for access control. In E. Damiani and P. Liu, editors, *DBSec*, volume 4127 of *Lecture Notes in Computer Science*, pages 179–193. Springer, 2006.

[21] K. Barthelmann. How to construct a hyperedge replacement system for a context-free set of hypergraphs. Technical report, Universität Mainz, Institut für Informatik, 1996.

[22] G. Bonfante, B. Guillaume, M. Morey, and G. Perrier. Modular Graph Rewriting to Compute Semantics. In J. Bos and S. Pulman, editors, *9th International Conference on Computational Semantics - IWCS 2011*, pages 65–74, Oxford, United Kingdom, Jan. 2011.

[23] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, Reading Mass, 1999.

[24] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An overview of ELAN. *Electr. Notes Theor. Comput. Sci.*, 15:55–70, 1998.

[25] T. Bourdier, H. Cirstea, D. J. Dougherty, and H. Kirchner. Extensional and intensional strategies. In *Proceedings Ninth International Workshop on Reduction Strategies in Rewriting and Programming*, volume 15 of *EPTCS*, pages 1–19, 2009.

[26] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 2008. Special issue on Experimental Systems and Tools.

[27] D. Clark, C. Hankin, and S. Hunt. Safety of strictness analysis via term graph rewriting. In J. Palsberg, editor, *SAS*, volume 1824 of *Lecture Notes in Computer Science*, pages 95–114. Springer, 2000.

[28] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(10):1367–1372, 2004.

[29] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation - part i: Basic concepts and double pushout approach. In *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*, pages 163–246. World Scientific, 1997.

[30] B. Courcelle. Graph Rewriting: An Algebraic and Logic Approach. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 193–242. Elsevier and MIT Press, 1990.

[31] V. Danos, J. Feret, W. Fontana, R. Harmer, and J. Krivine. Rule-based modelling of cellular signalling. In L. Caires and V. T. Vasconcelos, editors, *CONCUR*, volume 4703 of *Lecture Notes in Computer Science*, pages 17–41. Springer, 2007.

[32] V. Danos and C. Laneve. Graphs for core molecular biology. In C. Priami, editor, *CMSB*, volume 2602 of *Lecture Notes in Computer Science*, pages 34–46. Springer, 2003.

[33] M. Dauchet. Simulation of Turing machines by a left-linear rewrite rule. In *Proc. of RTA'89*, volume 355 of *Lecture Notes in Computer Science*, pages 109–120. Springer, 1989.

[34] F. Drewes, A. Habel, and H.-J. Kreowski. Hyperedge replacement graph grammars. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. 1: Foundations*, chapter 2, pages 95–162. World Scientific, 1997.

[35] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1-3.* World Scientific, 1997.

[36] C. Ermel, M. Rudolf, and G. Taentzer. The AGG approach: Language and environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 2: Applications, Languages, and Tools*, pages 551–603. World Scientific, 1997.

[37] M. Fernández. *Models of Computation.* Undergraduate Topics in Computer Science. Springer London, 2009.

[38] M. Fernández and I. Mackie. A calculus for interaction nets. In *Proceedings of PPDP'99, Paris*, number 1702 in Lecture Notes in Computer Science. Springer, 1999.

[39] M. Fernández and O. Namet. Graph creation, visualisation and transformation. In I. Mackie and A. M. Moreira, editors, *RULE*, volume 21 of *EPTCS*, pages 1–11, 2009.

[40] M. Fernández and O. Namet. Strategic programming on graph rewriting systems. In *Proceedings International Workshop on Strategies in Rewriting, Proving, and Programming, IWS 2010*, volume 44 of *EPTCS*, pages 1–20, 2010.

[41] M. Fowler and K. Scott. *UML Distilled: A Brief Guide to the Standard Object Modeling Language.* Addison-Wesley, Reading, Mass, 2nd ed edition, 2000.

[42] C. Fox, M. Fernández, and S. Lappin. Lambda Calculus, Type Theory, and Natural Language II. *J. Log. Comput.*, 18(2):203, 2008.

[43] C. Fuss, C. Mosler, U. Ranger, and E. Schultchen. The jury is still out: A comparison of agg, fujaba, and progres. *ECEASST*, 6, 2007.

[44] R. Geiß, G. V. Batz, D. Grund, S. Hack, and A. Szalkowski. Grgen: A fast spo-based graph rewriting tool. In A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, and G. Rozenberg, editors, *ICGT*, volume 4178 of *Lecture Notes in Computer Science*, pages 383–397. Springer, 2006.

[45] G. Gonthier, M. Abadi, and J.-J. Lévy. The geometry of optimal lambda reduction. In *Proceedings of the 19th ACM Symposium on Principles of Programming Languages (POPL'92)*, pages 15–26. ACM Press, 1992.

[46] A. Habel, J. Müller, and D. Plump. Double-pushout graph transformation revisited. *Mathematical Structures in Computer Science*, 11(5):637–688, 2001.

[47] A. Habel and D. Plump. Computational completeness of programming languages based on graph transformation. In *Foundations of Software Science and Computation Structures, 4th International Conference, FOSSACS 2001, Proceedings*, volume 2030 of *Lecture Notes in Computer Science*, pages 230–245. Springer, 2001.

[48] M. Hanus. Curry: A multi-paradigm declarative language (system description). In *Twelfth Workshop Logic Programming, WLP'97, Munich*, 1997.

[49] S. L. P. Jones. *Haskell 98 language and libraries: the revised report.* Cambridge University Press, 2003.

[50] C. Kirchner, F. Kirchner, and H. Kirchner. Strategic computations and deductions. In *Reasoning in Simple Type Theory. Studies in Logic and the Foundations of Mathematics, vol.17*, pages 339–364. College Publications, 2008.

[51] J. W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems: Introduction and survey. *Theor. Comput. Sci.*, 121(1&2):279–308, 1993.

[52] Y. Lafont. Interaction nets. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 95–108. ACM Press, 1990.

[53] V. Lipets, N. Vanetik, and E. Gudes. Subsea: an efficient heuristic algorithm for subgraph isomorphism. *Data Min. Knowl. Discov.*, 19(3):320–350, 2009.

[54] S. Lucas. Strategies in programming languages today. *Electr. Notes Theor. Comput. Sci.*, 124(2):113–118, 2005.

[55] I. Mackie, 2001. Habilitation à diriger des recherches en informatique, Université de Paris 7.

[56] I. Mackie. Efficient λ-evaluation with interaction nets. In V. van Oostrom, editor, *Proceedings of the 15th International Conference on Rewriting Techniques and Applications (RTA'04)*, volume 3091 of *Lecture Notes in Computer Science*, pages 155–169. Springer-Verlag, 2004.

[57] G. Manning and D. Plump. The GP Programming System. *ECEASST*, 10, 2008.

[58] G. Manning and D. Plump. The York Abstract Machine. *Electr. Notes Theor. Comput. Sci.*, 211:231–240, 2008.

[59] N. Martí-Oliet, J. Meseguer, and A. Verdejo. Towards a strategy language for Maude. *Electr. Notes Theor. Comput. Sci.*, 117:417–441, 2005.

[60] S. B. Navathe and R. Elmasri. *Fundamentals of Database Systems (Sixth Edition).* Addison Wesley, 2010.

[61] U. Nickel, J. Niere, and A. Zündorf. The FUJABA environment. In *ICSE*, pages 742–745, 2000.

[62] H.-O. Peitgen, H. Jürgens, and D. Saupe. *Chaos and Fractals.* Springer, Feb. 2004.

[63] G. Perez, I. Olmos, and J. A. Gonzalez. Subgraph isomorphism detection with support for continuous labels. In *FLAIRS Conference*, 2010.

[64] M. J. Plasmeijer and M. C. J. D. van Eekelen. *Functional Programming and Parallel Graph Rewriting.* Addison-Wesley, 1993.

[65] G. D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.

[66] D. Plump. Term graph rewriting. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 2: Applications, Languages, and Tools*, pages 3–61. World Scientific, 1998.

[67] D. Plump. The Graph Programming Language GP. In S. Bozapalidis and G. Rahonis, editors, *CAI*, volume 5725 of *LNCS*, pages 99–122. Springer, 2009.

[68] A. Rensink. The GROOVE Simulator: A Tool for State Space Generation. In *AGTIVE*, volume 3062 of *LNCS*, pages 479–485. Springer, 2003.

[69] A. Schürr, A. J. Winter, and A. Zündorf. The PROGRES Approach: Language and Environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 2: Applications, Languages, and Tools*, pages 479–546. World Scientific, 1997.

[70] M. Segal and K. Akeley. The OpenGL Graphics System: A Specification. Technical report, Silicon Graphics Inc., Dec. 2006.

[71] G. Taentzer, E. Biermann, D. Bisztray, B. Bohnet, I. Boneva, A. Boronat, L. Geiger, R. Geiß, A. Horvath, O. Kniemeyer, T. Mens, B. Ness, D. Plump, and T. Vajk. Applications of graph transformations with industrial relevance. chapter Generation of Sierpinski Triangles: A Case Study for Graph Transformation Tools, pages 514–539. Springer-Verlag, Berlin, Heidelberg, 2008.

[72] Terese. *Term Rewriting Systems*. Cambridge University Press, 2003. M. Bezem, J. W. Klop and R. de Vrijer, eds.

[73] R. Thiemann, C. Sternagel, J. Giesl, and P. Schneider-Kamp. Loops under strategies ... continued. In *Proceedings International Workshop on Strategies in Rewriting, Proving, and Programming*, volume 44 of *EPTCS*, pages 51–65, 2010.

[74] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, Jan. 1976.

[75] E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In *Proc. of RTA'01*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, 2001.

[76] E. Visser. A survey of strategies in rule-based program transformation systems. *J. Symb. Comput.*, 40(1):831–873, 2005.

Appendix A

Flag Sorting Example

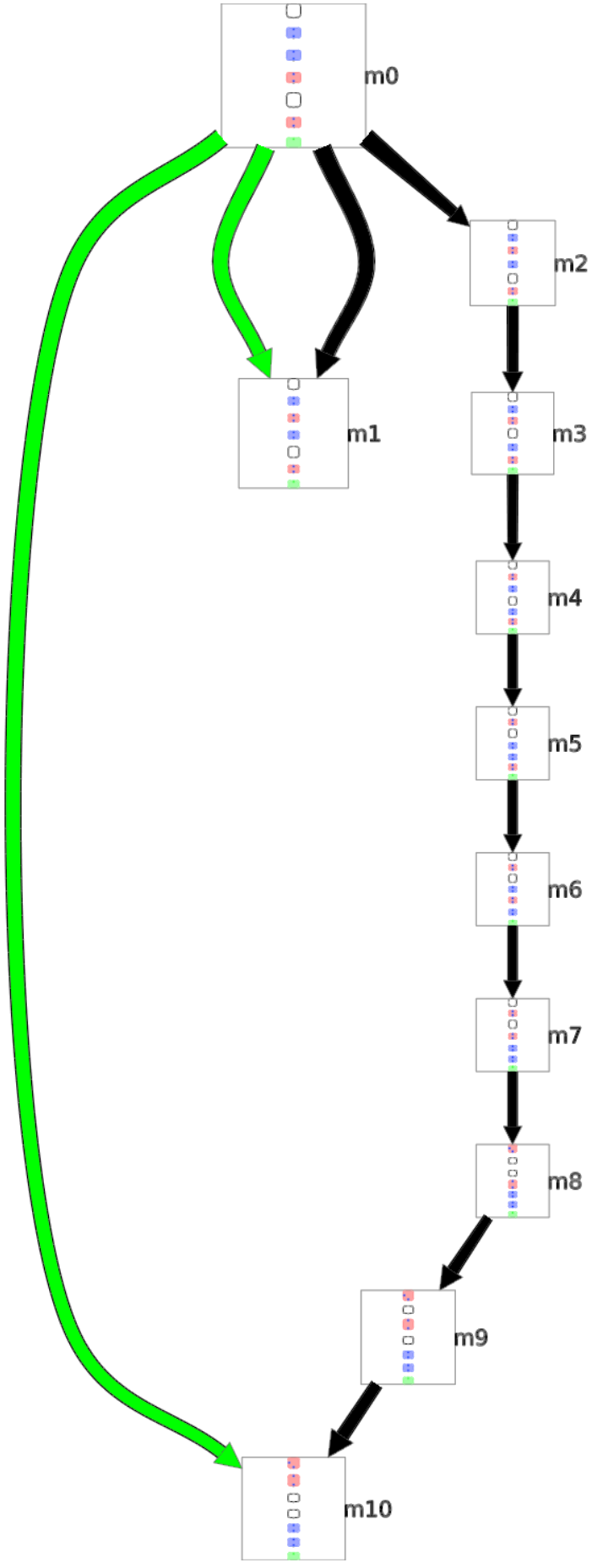


Figure A.1: A trace tree for the Flag Sorting example.

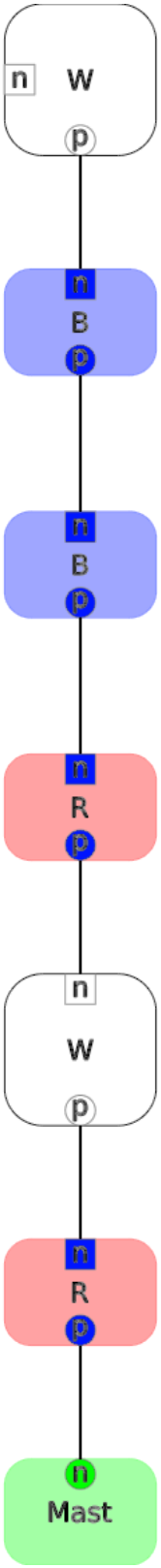


Figure A.2: The starting model for the Flag Sorting example, zoomed in to see port naming.

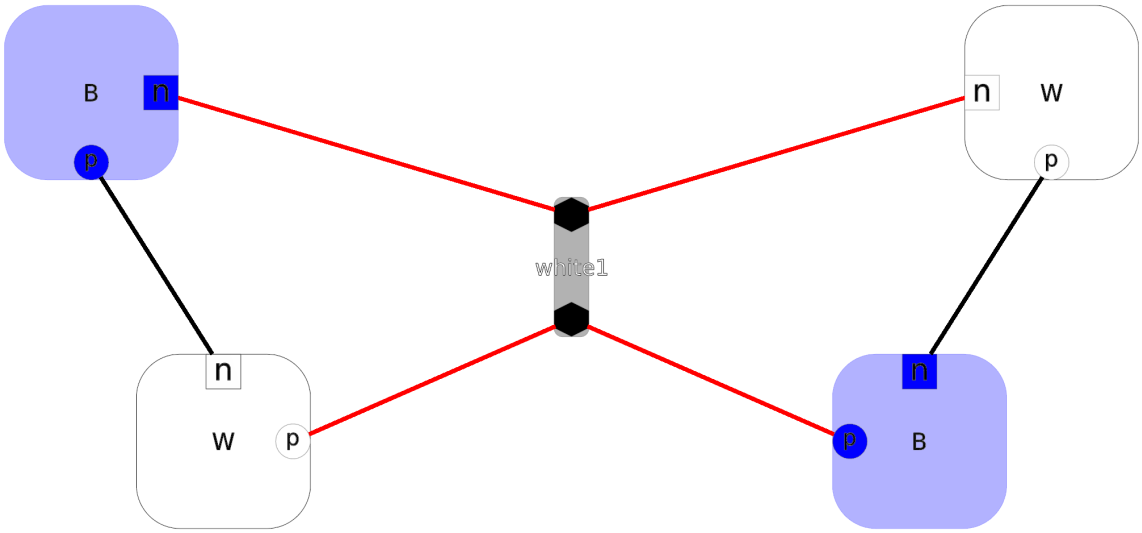


Figure A.3: The *white1* rule.

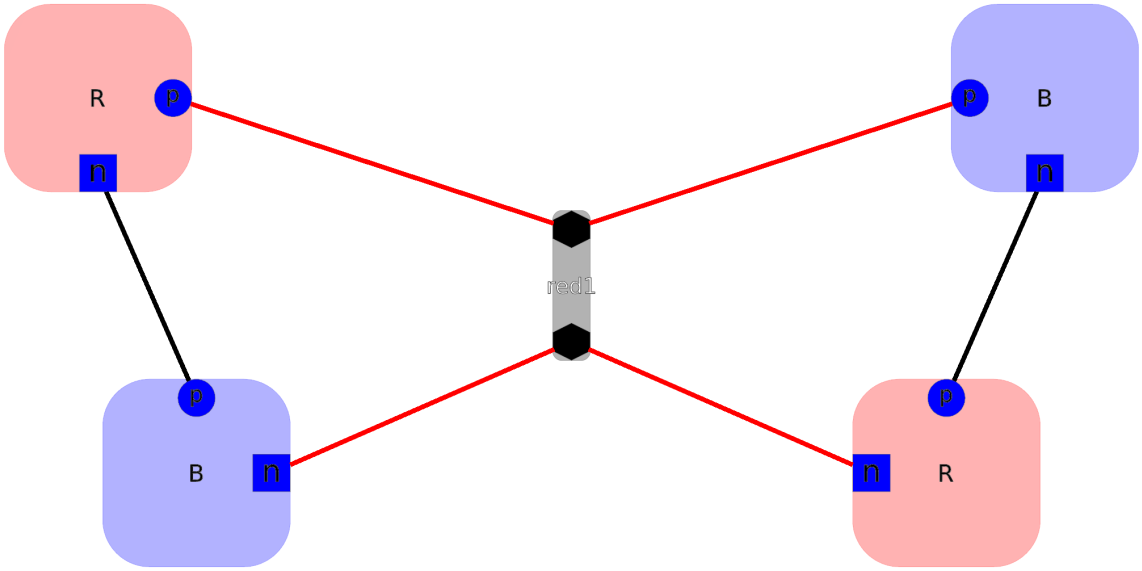


Figure A.4: The *red1* rule.

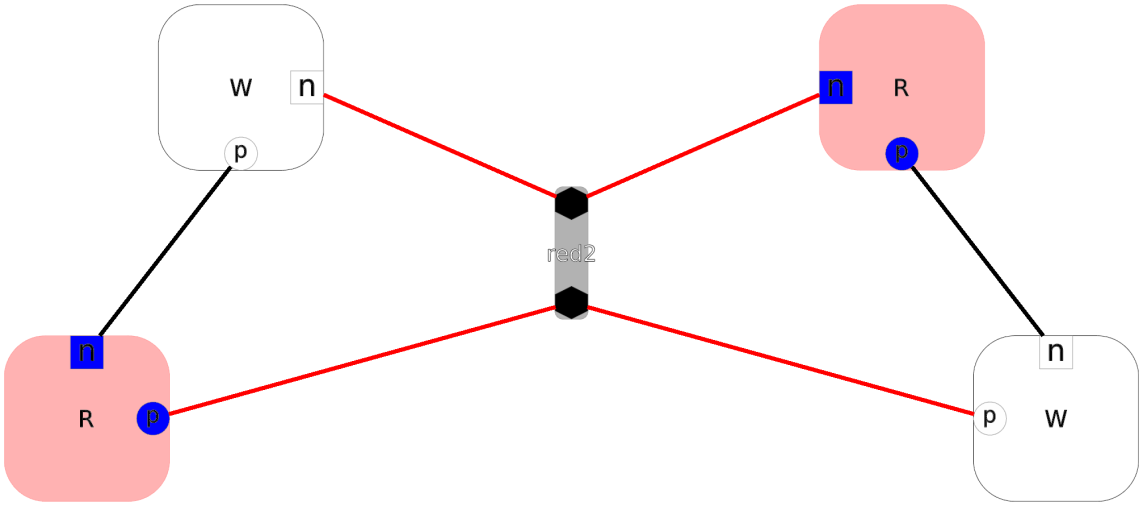


Figure A.5: The *red2* rule.

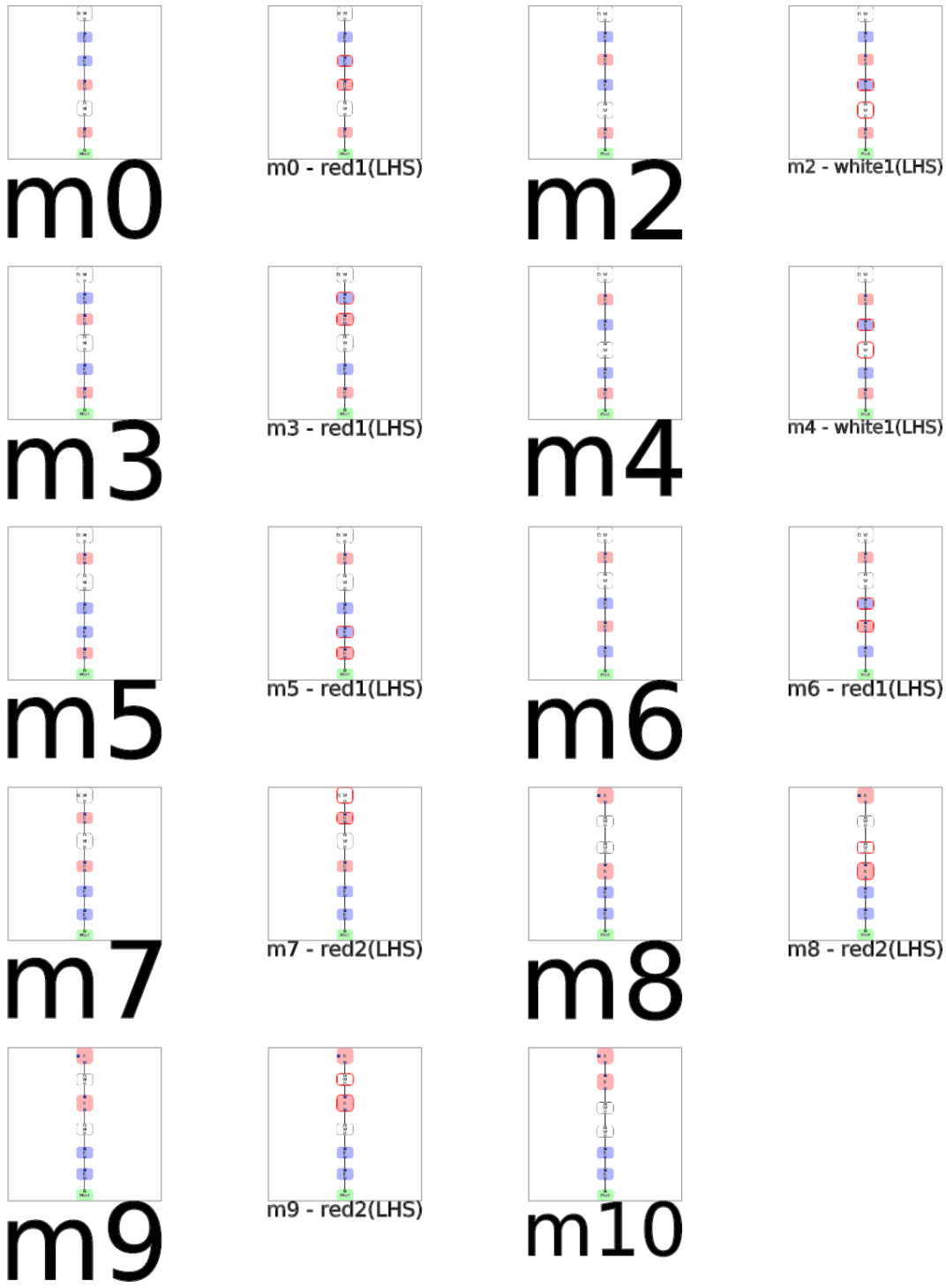


Figure A.6: A small multiples view for the Flag Sorting example: the large font means explicit models whereas the small font labels intermediate models that highlight the left hand side instance of a rule.

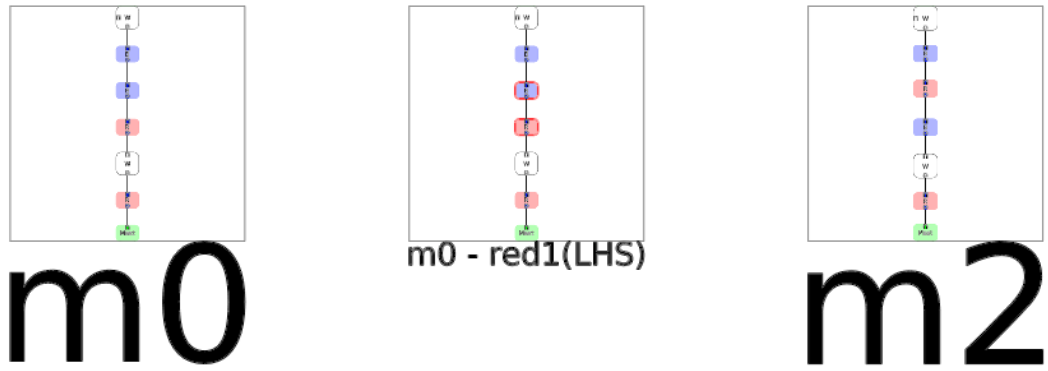


Figure A.7: A small multiples view for the Flag Sorting example zoomed in to the first line.

Appendix B

PORGY Tool

Figure B.1: PORGY node, rule and graph creation main window.

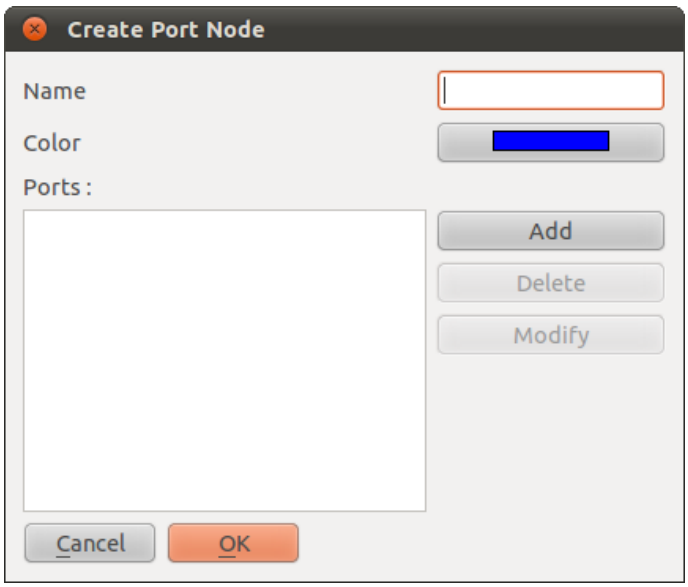


Figure B.2: The *add node* window.

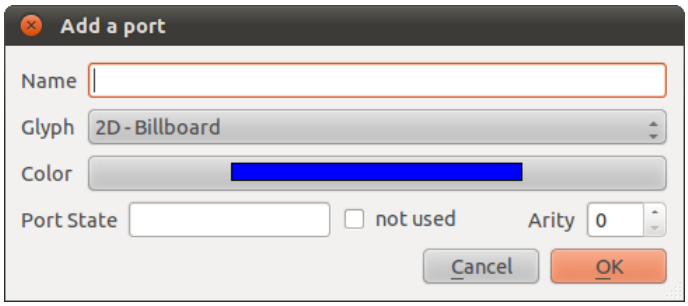


Figure B.3: The *add port* window.

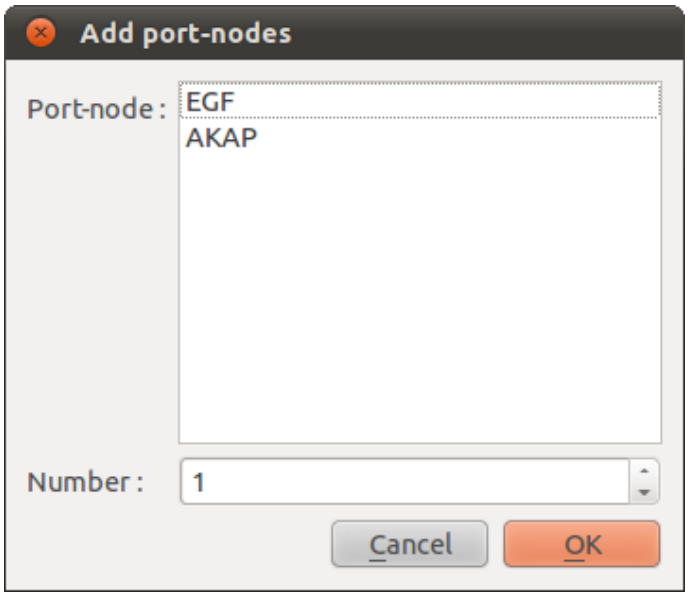


Figure B.4: The *add nodes to a model* window.

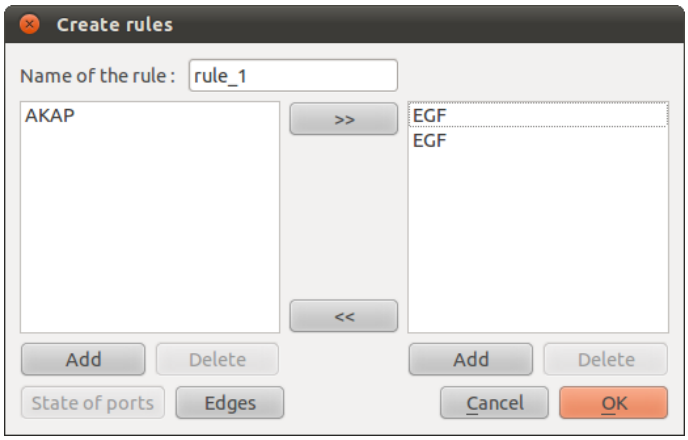


Figure B.5: The rule creation window.

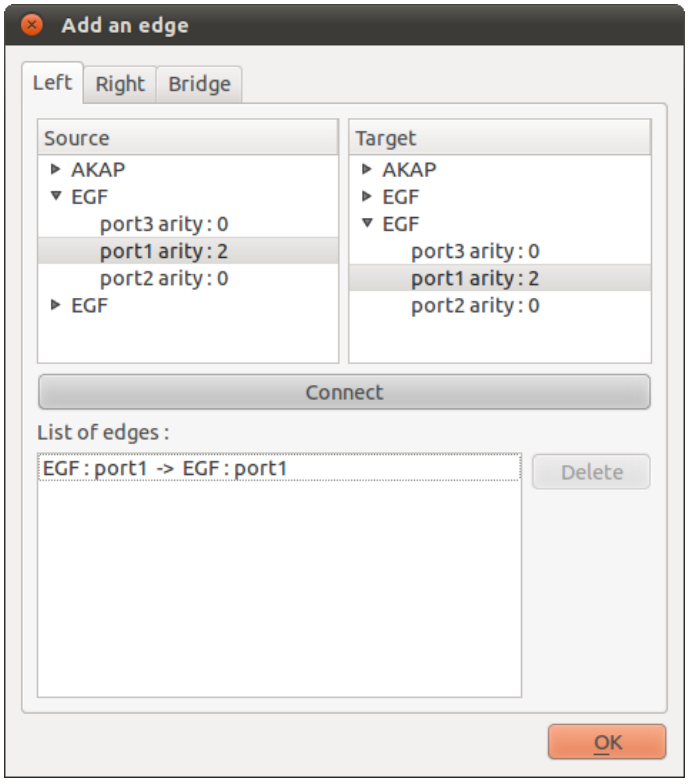


Figure B.6: The *add edge* window for rules (Left wiring).

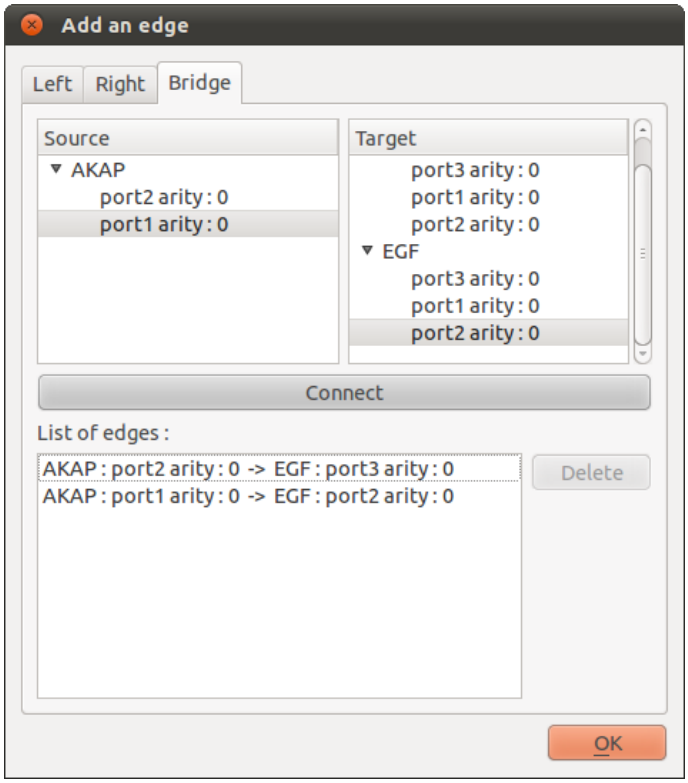


Figure B.7: The *add edge* window for rules (Bridge wiring).

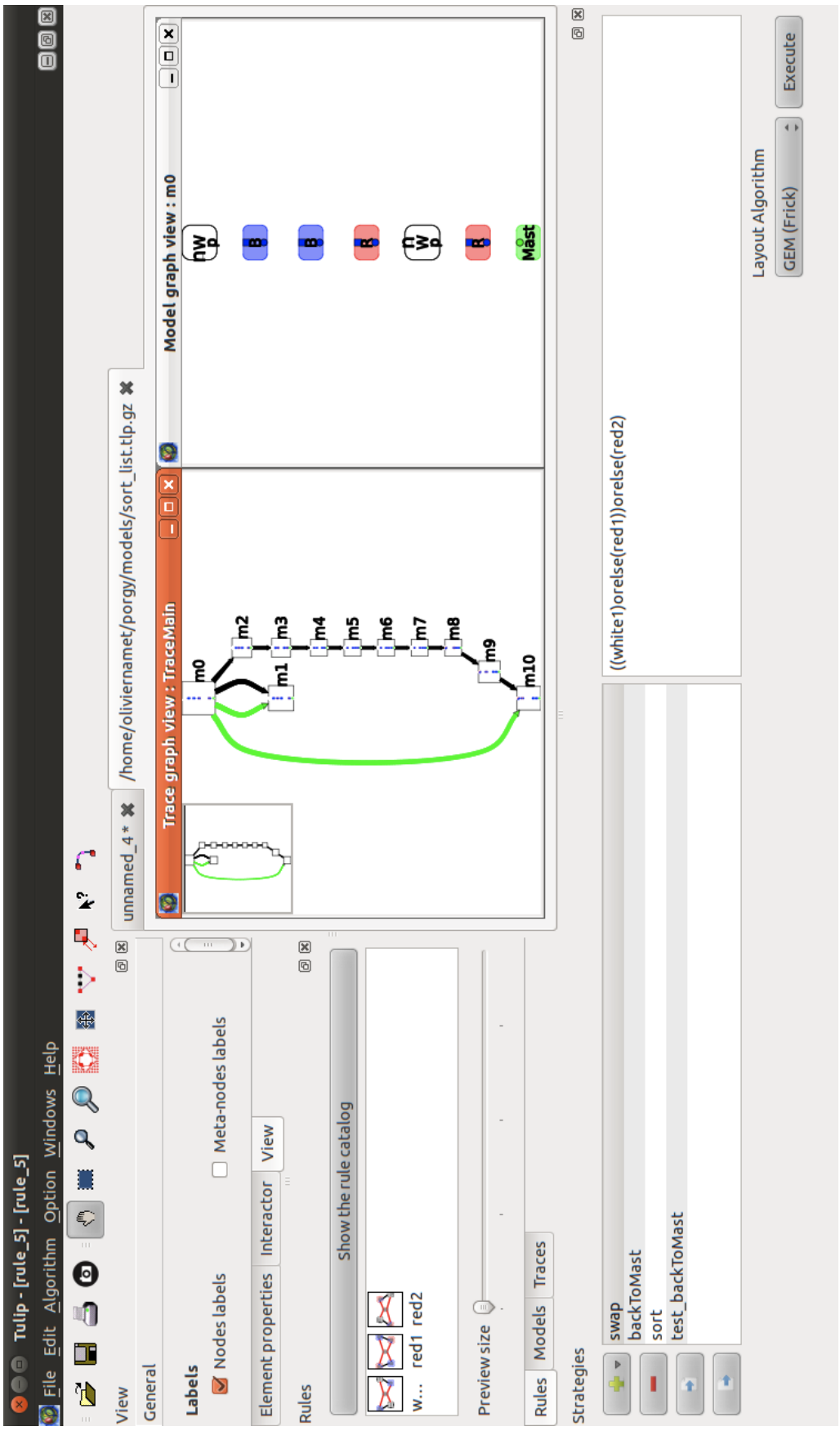


Figure B.8: The main PORGY window.

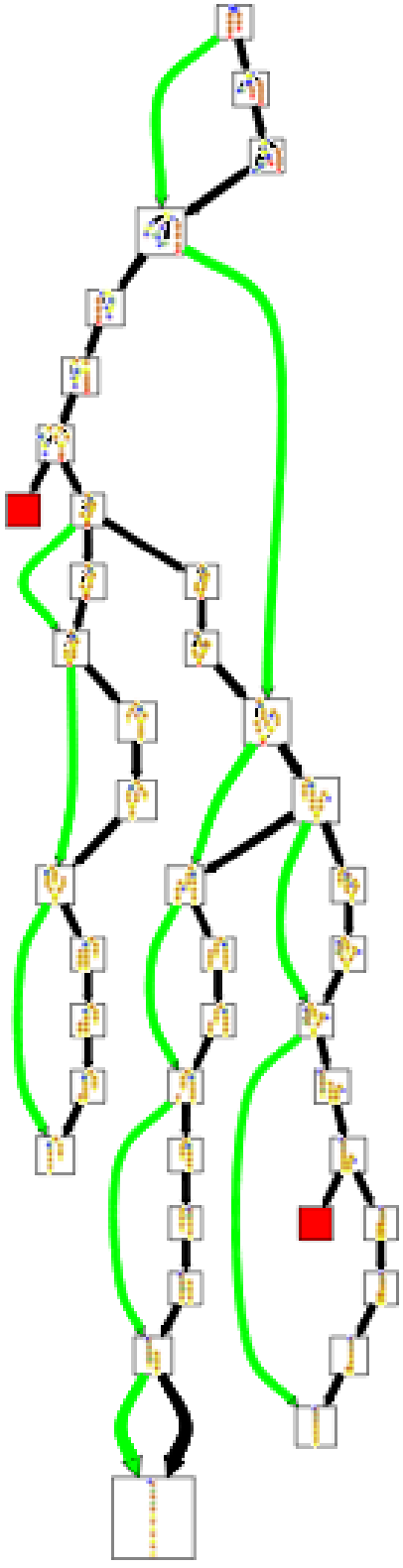


Figure B.9: An example trace in PORGY (including some Fail nodes).

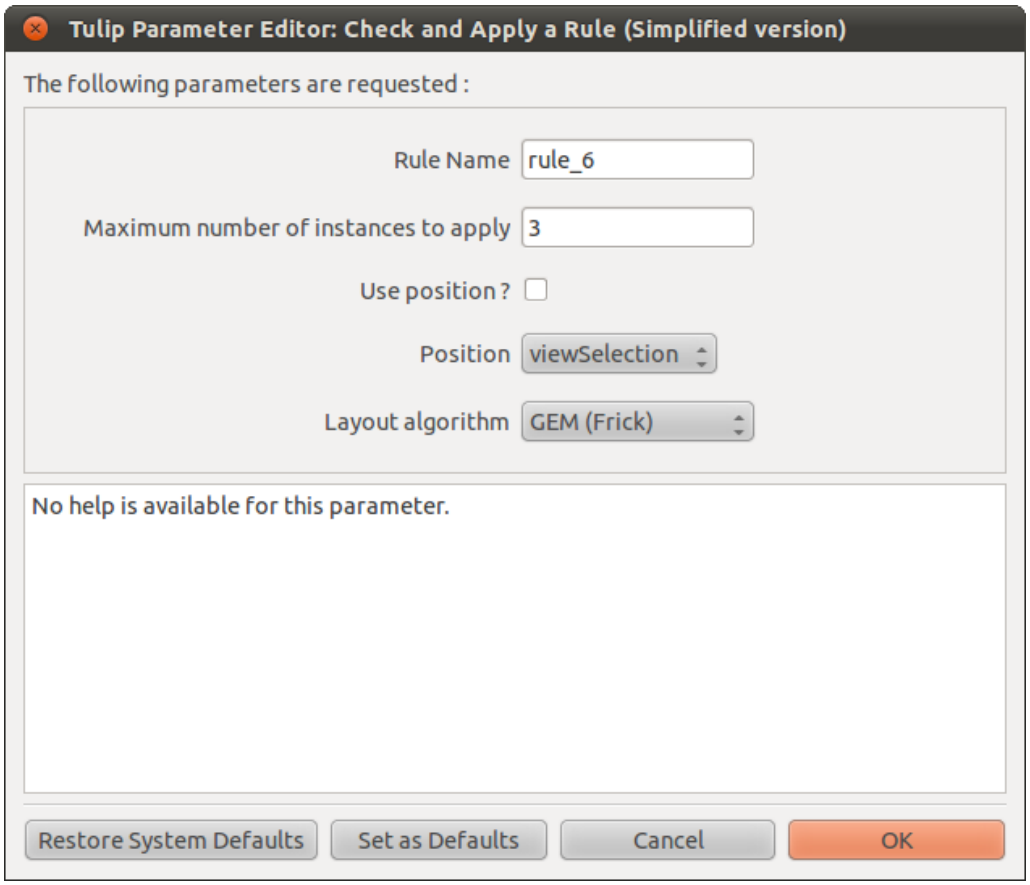


Figure B.10: The apply rule window.

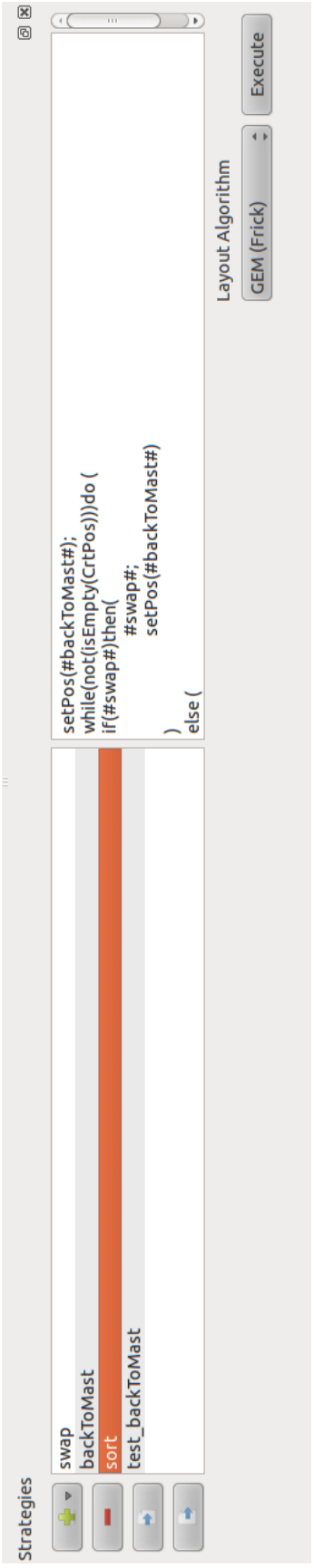


Figure B.11: The strategy panel.

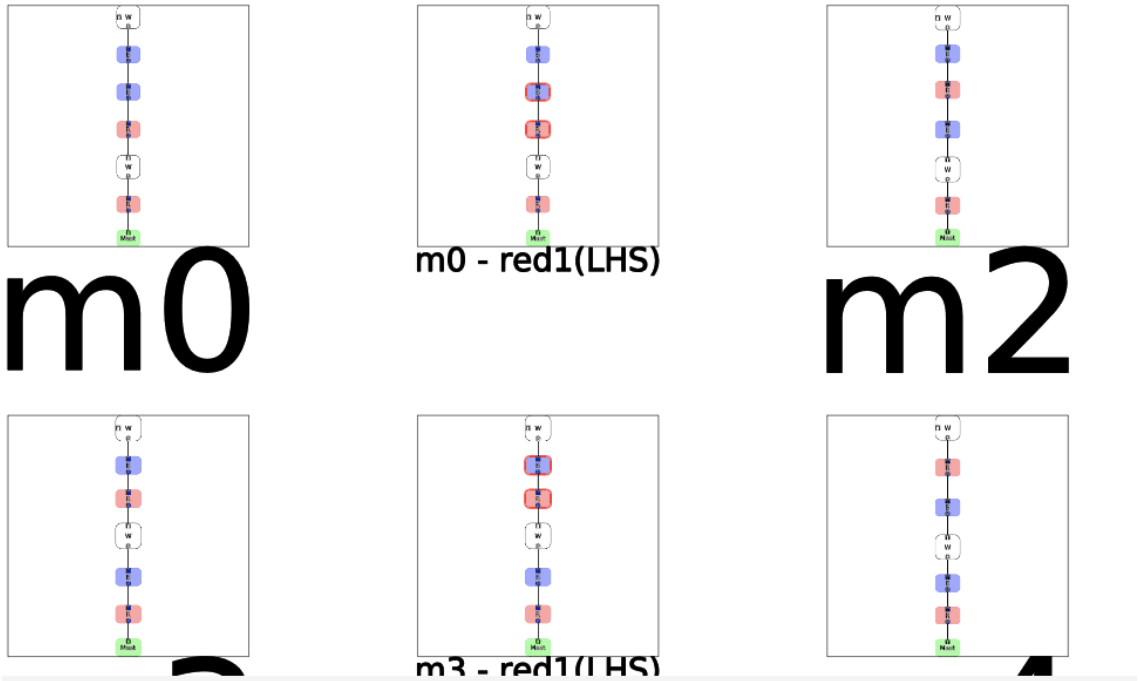


Figure B.12: The small multiples view panel..

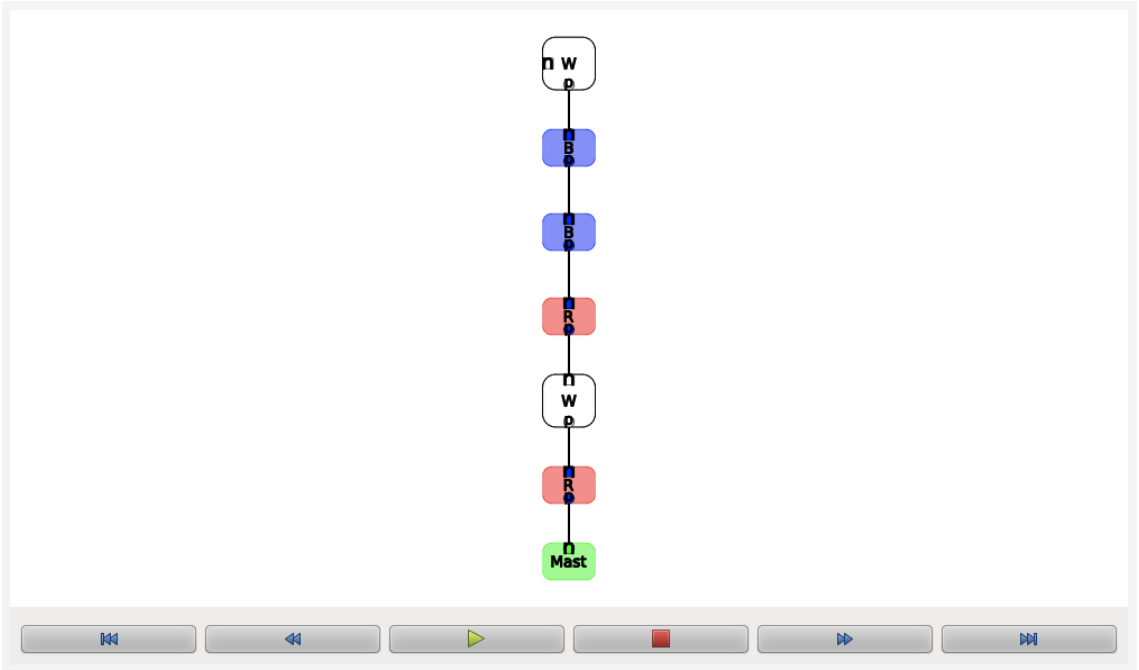


Figure B.13: The animation panel.