

# Bigraph-Based Taxonomy Framework

Master's Thesis  
by

Rupa Gadiraju  
Martin Nørskov Jensen  
Raghava Rao Mukkamala

Supervisor: Thomas Hildebrandt

IT University of Copenhagen  
September 1, 2005

## **Abstract**

In this thesis we design and implement a bigraph-based framework for working with taxonomies.

We define a set of requirements for the framework, and analyze to what degree a framework based on bigraphs will satisfy these requirements. Realizing the existing bigraph theory alone cannot satisfy the requirements, we present a set of formalized extensions to the theory.

We present an analysis of the framework architecture in which we discuss how the definitions can be implemented in an object oriented programming environment.

Finally we discuss how successful our framework implementation has been, and what advantages and disadvantages basing the framework on bigraphs gave us. Realizing that current bigraph theory has some shortcomings concerning practical implementation, we offer some suggestions as to how this can be improved.

# Contents

<b>1</b>	<b>Project Description</b>	<b>4</b>
1.1	Background . . . . .	4
1.1.1	The Taxonomy Tool . . . . .	4
1.1.2	Bigraphs . . . . .	5
1.2	Problem definition and methodology . . . . .	6
1.3	Report . . . . .	7
1.4	Acknowledgments . . . . .	7
<b>2</b>	<b>Framework Requirements</b>	<b>8</b>
2.1	The Taxonomy Tool . . . . .	8
2.2	Use Cases . . . . .	9
2.2.1	Hierarchical object structures . . . . .	9
2.2.2	Ontologies . . . . .	10
2.2.3	Workflows . . . . .	10
2.3	Design Goals . . . . .	12
2.3.1	Practical and efficient . . . . .	12
2.3.2	Persistent taxonomy of objects . . . . .	12
2.3.3	Relations and Mappings . . . . .	13
2.3.4	Accessibility and compatibility . . . . .	13
2.4	Requirements . . . . .	13
2.5	Summary . . . . .	15
<b>3</b>	<b>Bigraphs</b>	<b>16</b>
3.1	A short introduction to bigraphs . . . . .	16
3.1.1	Background . . . . .	16
3.1.2	Concepts . . . . .	16
3.1.3	Bigraph example: workflow . . . . .	18
3.2	Bigraphs defined . . . . .	19
3.2.1	Definitions . . . . .	19
3.3	Bigraph building blocks . . . . .	24
3.3.1	Placings . . . . .	24
3.3.2	Wirings . . . . .	25
3.3.3	Discrete ions . . . . .	25
3.3.4	Building blocks example . . . . .	25
3.4	Summary . . . . .	26

<b>4</b>	<b>Data Structure Analysis</b>	<b>30</b>
4.1	Analysis background . . . . .	30
4.2	XML . . . . .	31
4.3	OWL . . . . .	31
4.3.1	OWL-based use case implementations . . . . .	32
4.3.2	Summary . . . . .	37
4.4	Bigraphs . . . . .	37
4.4.1	Bigraph-based use case implementations . . . . .	37
4.4.2	Summary . . . . .	39
4.5	Data structure comparison . . . . .	39
4.6	Extending bigraphs . . . . .	40
4.6.1	Bigraph element names . . . . .	40
4.6.2	Types . . . . .	40
4.6.3	Node properties . . . . .	41
4.6.4	External references . . . . .	41
4.6.5	Persistence and sharing . . . . .	42
4.6.6	Extension formalization . . . . .	43
4.7	Summary . . . . .	45
<b>5</b>	<b>Framework Architecture Analysis</b>	<b>46</b>
5.1	Overall framework structure . . . . .	46
5.1.1	Framework structure . . . . .	46
5.2	Framework core . . . . .	48
5.2.1	Base bigraphs . . . . .	50
5.2.2	Extended bigraphs . . . . .	60
5.3	Data access layer . . . . .	64
5.4	Data layer . . . . .	64
5.4.1	Relational database implementation . . . . .	65
5.4.2	XML implementation . . . . .	65
5.5	Summary . . . . .	66
<b>6</b>	<b>Implementation</b>	<b>67</b>
6.1	Framework structure . . . . .	67
6.2	Framework implementation . . . . .	67
6.2.1	Base bigraph implementation . . . . .	68
6.2.2	Extended bigraph implementation . . . . .	74
6.2.3	Data access layer . . . . .	79
6.2.4	Data layer . . . . .	81
6.3	Summary . . . . .	82
<b>7</b>	<b>User Guide</b>	<b>83</b>
7.1	Creating a bigraph . . . . .	83
7.2	Linking nodes . . . . .	87
7.3	Persistence . . . . .	88
7.4	Composition . . . . .	88
7.5	Composite bigraph . . . . .	89
7.6	Summary . . . . .	90

<b>8</b>	<b>Test</b>	<b>91</b>
8.1	Test method . . . . .	91
8.2	Test cases . . . . .	91
8.2.1	Builder methods . . . . .	91
8.2.2	Composition . . . . .	93
8.2.3	Composite bigraphs . . . . .	93
8.2.4	Data access . . . . .	94
8.3	Test results . . . . .	94
8.4	Summary . . . . .	95
<b>9</b>	<b>Conclusion</b>	<b>96</b>
9.1	Conclusion . . . . .	96
9.1.1	Bigraph evaluation . . . . .	97
9.2	Perspectives . . . . .	98
9.2.1	Taxonomy framework . . . . .	98
9.2.2	Bigraphs . . . . .	99

# Chapter 1

## Project Description

In this chapter we will describe the background of the project, and based on this we will present a proper problem definition.

### 1.1 Background

The original motivation for this project is the stated necessity for a new version of the Taxonomy Tool, a software application owned by the Danish software company Resultmaker A/S. All the project members are currently employed by Resultmaker.

#### 1.1.1 The Taxonomy Tool

As its name indicates, the Taxonomy Tool is a tool for working with taxonomies, collections of objects organized as trees. With this tool the user can create taxonomies of different kinds of objects and create relations (or links) between these objects. These taxonomies are persisted, and several users can work with the same taxonomies simultaneously. The original Taxonomy Tool is described in more detail in chapter 2.

This basic functionality has turned out to be useful in a number of scenarios.

For example, the current Taxonomy Tool has been employed in online workflow creation projects. Online workflows are used when creating digital forms. One workflow can incorporate several forms, and in this way the user will not have to input the same data more than once. The workflow guides the user through the forms, and the structure of the workflow may depend on the input of the user. The workflows were constructed in the Taxonomy Tool because of their tree structures.

The Taxonomy Tool has also been employed in categorization projects, e.g. for search engine meta data. By coupling each form in a taxonomy of government forms to a keyword in an ontology of keywords, it is possible to generate keywords for each form.

Organization of data for analysis projects is another key application domain for the Taxonomy Tool. By coupling above mentioned taxonomy of forms to other taxonomies with external references to statistical data, statistical analyses can be generated for the forms.

Seen from a larger perspective, tree structures or taxonomies are used in many different applications and fields. For example, the Semantic Web project<sup>1</sup> is an effort to create a data model for the exchange of data between machines on the Internet. Data is organized into ontologies structured as trees, i.e. taxonomies.

While thinking about the design of the new Taxonomy Tool, we decided that it would not be sufficient to just build a new tool. We wanted to build a complete framework for working with taxonomies, so that persisted taxonomies can easily be accessed programmatically. This will make it possible to make different kinds of tools for working with this very flexible data structure in different ways.

After deciding on the design and development of a taxonomy framework as the subject of our thesis, we came across the notion of bigraphs.

### 1.1.2 Bigraphs

A bigraph is a data structure that incorporates several trees and allows for relations (links) between two or more nodes in the trees. So far, bigraphs have mainly been employed in models of process calculus and mobile computation.[3] Bigraphs are described in more detail in chapter 3.

The theory of bigraphs is based on a well developed formal mathematical model that also includes definitions of various operations on bigraphs, e.g. composition of bigraphs.

Bigraphs are suitable for modeling the kinds of structures employed by a taxonomy/ontology framework due to their inherent hierarchical structures. Additionally, bigraph theory defines links between nodes, which makes bigraphs even more suitable for the taxonomy framework we want to make. Also, the compositionality of bigraphs makes it possible to add useful features, e.g. sharing of structures, that would otherwise not be included.

Because bigraphs are structurally similar to the taxonomies of the taxonomy framework and because they will allow us to add interesting features such as compositionality, we have decided to build a prototype of the taxonomy framework utilizing bigraphs as the internal data structure. The purpose of this is twofold: we want to examine how well suited bigraphs are for the taxonomy framework specifically, and we also want to examine how well suited bigraphs are as a data structure for applications generally.

As far as we know, there does not currently exist a software implementation of bigraphs that can be used in the context of the framework. Therefore, a substantial part of this thesis will be to develop such an implementation.

We want our bigraph implementation to be as faithful to the existing bigraph theory as possible. This is relevant for the naming of components, as well as for the ways components function and the restrictions they have.

Although we consider bigraphs suitable for the taxonomy framework, they were invented with other purposes in mind, and certain features of bigraphs make them difficult to use in such a context if employed without modifications. Therefore we have decided to implement a framework for working with bigraphs as defined by the existing bigraph theory, and then extend that to make it more suitable for the taxonomy framework. We will aim at designing all extensions to be compatible with bigraph theory, in the sense that we will not extend

---

<sup>1</sup><http://www.w3.org/2001/sw/Activity>

the theory beyond recognition and we will provide formal definitions of the extensions.

## 1.2 Problem definition and methodology

The purpose of this thesis is to examine how well suited bigraphs are as a) the internal data structure of the taxonomy framework and b) a data structure for applications in general. We will do this by designing and developing a prototype of a bigraph-based framework for working with taxonomies.

We will analyze and define the requirements and specifications for this framework. These will be based on the current Taxonomy Tool and a number of use cases derived from actual projects in which the Taxonomy Tool has been utilized. We will analyze what improvements to the current application we deem necessary or useful based on these use cases, and these will be included in the specified requirements.

We will present the formal mathematical theory on which bigraph theory is based. This will provide the basis for the design of the framework.

We will perform a survey of data structures that could be used as the internal data structure of the framework, including bigraphs. This we will do in order to gain inspiration from various sources, but also to discuss how suitable bigraphs are for our purposes compared to other similar data structures.

Based on the defined requirements and the data structure survey we will discuss what extensions will need to be made to the defined bigraph theory in order to fulfill the requirements of the framework and we will formalize these extensions.

We will design an implementation of the presented bigraph theory. This we will extend to provide a bigraph implementation that is suitable for the taxonomy framework.

We will design a taxonomy framework that is based on above mentioned bigraph implementation and consists of a logic layer with a well defined application interface (API) and a data access layer (DAL) for accessing and persisting data.

We will present an implementation of the framework based on the preceding design. We will implement as much as we consider feasible within the context of this project. This will as a minimum include a functional logic layer and data access layer as well as a small application for viewing taxonomies, built by utilizing the framework.

Developing a complete framework and tools that utilize the framework will be necessary to realize the full potential of the framework that we design. However, this would be a massive task that we consider to be outside the scope of this thesis. We will implement a prototype of the framework, with enough functionality to function as a proof of concept. This means that all of the features described as required will not necessarily be included in the implementation.

We will evaluate the framework prototype. We will concentrate on evaluating what advantages and disadvantages employing bigraphs as the internal data structure gave us. We will also discuss how well suited bigraphs are as a data structure for applications in general, and in this context we will draw comparison to similar data structure implementations.

### 1.3 Report

In chapter 2 we will present an introduction to the current Taxonomy Tool, and we will offer a couple of use cases based on projects in which the Taxonomy Tool has been employed. We will then define four major design goals, and based on these we will define a number of requirements for the taxonomy framework.

In chapter 3 we will present a brief introduction to bigraphs. We will provide formal mathematical definitions of all the components we will be utilizing in this thesis. We will also describe bigraph building blocks, which are base bigraphs out of which all bigraphs can be constructed.

In chapter 4 we will compare bigraphs to other similar data structures to get an idea of how well suited bigraphs are for the taxonomy framework. Based on this we will present a number of extensions to the existing theory, including formal definitions of these extensions.

In chapter 5 we will provide an analysis of the framework architecture. We will discuss how the framework can be implemented based on the definitions provided in chapters 3 and 4.

In chapter 6 we will present a description of the implemented taxonomy framework.

In chapter 8 we will present the test strategy for the framework implementation including the results of running the tests.

Finally, in chapter 9, we will present the conclusions we have reached during the course of writing this thesis, and offer some perspectives for future work.

The appendix section of this thesis contains the source code of the framework implementation and test output.

The reader of this report would benefit from some familiarity with formal mathematical definitions, although this is not required. In particular, people familiar with bigraph theory should have no problems reading the more mathematically inclined chapters.

All code written in the context of this thesis will be written in C# and a familiarity with this programming languages will be an advantage for the reader in certain chapters, although this is not a requirement.

### 1.4 Acknowledgments

We would like to thank our supervisor Thomas Hildebrandt for supreme patience when introducing us to the realm of bigraphs in particular and for help on writing the thesis report in general.

## Chapter 2

# Framework Requirements

In this chapter we will define the requirements of the framework. This we will do by first examining the existing Taxonomy Tool and describing some use cases for the application. Then we will define a number of basic design goals and from these we will derive a set of concrete requirements.

### 2.1 The Taxonomy Tool

The Taxonomy Tool derives its name from its own internal data structure. Data is organized as taxonomies or what in computer science is commonly referred to as trees. Nodes in the trees can have properties, but most non-structure related data is not stored in the tool itself. All nodes are typed, and the type definition coupled with an external ID reference specifies where the data of a node is stored and how it is accessed.

For example, the current Taxonomy Tool contains a taxonomy of all business related forms employed by the Danish government. The taxonomy is organized so that each form is a child of the authority that administrates that form (there are no restrictions regarding parent-child relationships between nodes of different types in the current tool). But the data pertaining to each form, such as form name, status, link information etc., is actually stored in a separate database, although it can still be accessed through the Taxonomy Tool.

Because non-structural data is stored in a separate database, it is easy to build taxonomies based on already existing data, although no standardized import functionality exists for this. It is simply a matter of defining the rules for establishing the relationships, and then writing code that generates a node for each entity in the database, with a reference to the original entity.

This separation of structural data and type specific (non-structural) data makes the Taxonomy Tool a rather flexible tool, since any kind of object can perceivably be represented in a taxonomy. We consider this kind of flexibility to be a major design goal in this project as well.

One limitation in the current implementation, is that nodes can only be externally linked to data in databases, and the databases must be accessed directly. It would be useful to also be able to reference data stored in other kinds of data sources, such as XML documents, or data that is only accessible through other applications.

What the current Taxonomy Tool lacks is the specification of rules, such as restrictions on what types of nodes can be related, a good security/authorization system, import/export functionality and a programming interface (API).

## 2.2 Use Cases

In this section we will describe three use cases, that are all based on actual projects in which the current Taxonomy Tool has been employed.

These use cases will be referred to continually throughout the whole report.

### 2.2.1 Hierarchical object structures

The most basic use case is that of building a hierarchy (or taxonomy) of objects based on already existing data. This is useful when the existing data has a flat or no structure but the objects that the data represents has a hierarchical structure.

An example of an existing taxonomy like this is the taxonomy of business-related government forms mentioned in section 2.1. In this taxonomy each form is the child of a government authority which is a child of a government ministry. The taxonomy contains external references to a database of forms, so that metadata for each form can be accessed.

This taxonomy becomes really useful when coupled with other taxonomies. For example, in one project each form was related to a law in a taxonomy of laws, based on what law necessitated that form (see fig. 2.1). Since the administrative burden of each law was known, it was then possible to calculate the approximate administrative burden of each form, and this made it possible to estimate how much money could be saved by providing each form as a digital form.

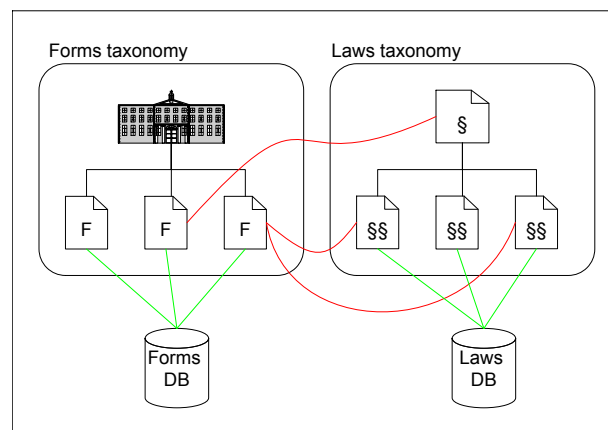


Figure 2.1: The forms taxonomy coupled with the laws taxonomy

This use case shows the usefulness of being able to externally reference data. Seemingly unrelated abstract objects (such as forms and laws) from different data stores and with different data structures can be related and useful information can be extracted based on this.

An issue regarding the current Taxonomy Tool is its lack of a good type system. Although nodes are typed, the type of a node only defines how external data are accessed. The type of nodes has no influence on the syntax of a taxonomy, i.e. how nodes are related vertically and horizontally. This means that it is easy to make mistakes when working with the tool, because the application does not detect syntax errors. It is, for instance, possible to have an authority node be the child of a form node as well as the other way around. And since horizontal relations are not typed at all, it is possible to relate any nodes horizontally. A better type system would be a big improvement over the old tool.

### 2.2.2 Ontologies

One particularly interesting feature of the current Taxonomy Tool is its ability to handle not only taxonomies, but also hierarchically structured ontologies. In this context we use the following definition of the word “ontology”:

An ontology defines the terms used to describe and represent an area of knowledge. Ontologies are used by people, databases, and applications that need to share domain information [...]. Ontologies include computer-usable definitions of basic concepts in the domain and the relationships among them [...]. They encode knowledge in a domain and also knowledge that spans domains. In this way they make knowledge reusable.[10]

An ontology, therefore, is simply a structured definition of a knowledge domain, and an ontology can take several different forms, e.g. as a taxonomy, a metadata scheme or a collection of logical theories. In the context of the Taxonomy Tool, this has of course always been as taxonomies. What is useful about ontologies, is that they make information understandable by machines and reusable in different contexts.

In the context of this framework, ontologies do not differ significantly from taxonomies of objects. The formal difference is that ontologies organize hierarchies of concepts or ideas whereas objects in taxonomies usually refer to concrete objects. As an illustrative example, we could say that an ontology would organize concepts such as fiction, non-fiction and travel (book genres) whereas a taxonomy of objects would organize actual books.

When it comes to the data structure of this framework, the distinction between taxonomies and ontologies loses its meaning. Therefore they shall be treated as the same in the context of the framework.

### 2.2.3 Workflows

In the context of this report we consider a workflow to be a series of online input pages, where the content of one input page may or may not be determined by the data input at previous pages. At the end of the workflow the input data is stored on a server.

For example, companies need to report information to authorities on a monthly and sometimes daily basis. It is becoming more and more common for this to be done through online workflows, which saves time by reuse of data and lowers the rate of input errors.

Fig. 2.2 shows a simple example of a workflow for reporting trade to tax authorities. First the user fills out information about the company such as address, company registration number etc. If the company has international trade the user then reports that after which domestic trade is reported. Otherwise the user just reports domestic trade. Finally the input data is summarized on the last page, including information on how much tax the company should pay.

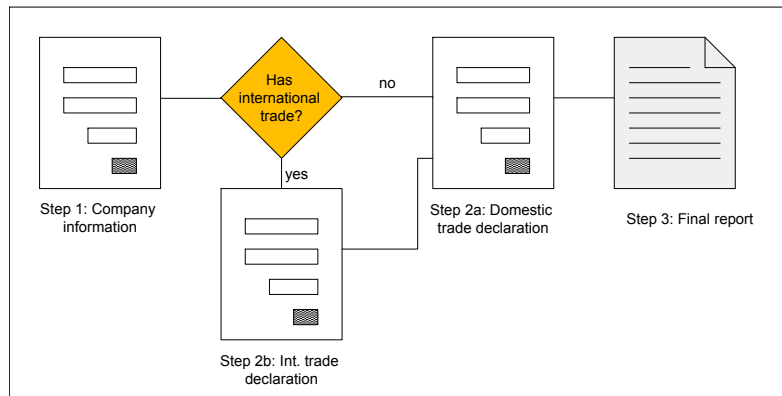


Figure 2.2: A simple workflow for reporting trade to tax authorities

Workflows are an example of how it is sometimes useful to be able to create taxonomies just for their hierarchical structure. In these cases it is not necessary to reference data from other data sources.

For example, workflows can be created as hierarchical structures, with each node representing a possible step in the process. The children of a node each represent a possible outcome of the choices the user will have to take at that node.

This use case also highlights the usefulness of reusing structures. Often the same workflow step is included in several workflows, e.g. a company address information input page. Instead of creating a new page for each workflow, it would be a lot more efficient to use the same template in all workflows. This also makes maintaining the workflows easier, because updates will only need to be done in one place when that template is changed.

When having nodes with no inherent data, it can be useful to be able to set properties on nodes. This can also be useful for nodes with inherent data, because the user might want to set properties that are specific to the taxonomy, and thus do not belong in an external data source.

This use case is related to the relatively new field of research on workflows. The existing research operates with a broader understanding of what a workflow is and thus the term in this context also includes e.g. work processes. van der Aalst et al. has published a paper describing 23 workflow patterns [7] such as sequences, splits and merges.

This research supports the notion of the usefulness of reusing structures. In the context of this research we imagine that each pattern be defined once and then instantiated whenever an instance is needed. This is quite similar to the case of reusing described above.

## 2.3 Design Goals

In this section we will first describe the general design goals for the taxonomy framework, and in the next section we will then define a number of requirements based on these design goals.

Some of the design goals have been inspired by the use cases defined for the RDF-based OWL (Web Ontology Language) language [10].

### 2.3.1 Practical and efficient

In the problem definition we have already defined that we want to create a framework for working with bigraphs programmatically. This implies the need for an application programming interface (API). Such an API should be practical to use for programmers, meaning it should not require extensive knowledge of the underlying data formats and it should be possible to build and manipulate taxonomies in relatively few lines of code.

With such an API it will be possible to create several taxonomy-based applications based on the same framework and even handling the same data.

The user should be able to do transformations on a taxonomy. This could range from simple changes, such as assigning a new parent node to a node in a taxonomy, to structure-wide changes such as completely rearranging all nodes based on some rule.

The framework should be efficient, meaning that operations should not require unreasonable amounts of resources. Taxonomies can be very large and complex, and the framework should scale to handle even large structures. In order to accomplish this, efficient algorithms should be employed for the implementation of potentially resource demanding operations.

To work efficiently with concrete taxonomies will probably require some kind of GUI. However, in this project we will concentrate on designing and implementing a framework with which such GUI's can be easily created. We choose to do this because creating a useful programming tool will make creating a GUI much easier. The creation of a GUI will be important in actually realizing the usefulness of the framework, but in the context of this project it is less interesting.

### 2.3.2 Persistent taxonomy of objects

The most basic activity a user will be doing with the taxonomy framework is to build a taxonomy, or tree, of objects. This taxonomy should be persistent, i.e. it should be possible to store it.

This taxonomy might be based on an already existing data source, such as a database of forms, in which case this external data source should be referenced for the data pertaining to individual nodes. The alternative would be to copy data from the external data source to the taxonomy, but this would mean that updates to the original data would not be pushed through to the taxonomy, and eventually the taxonomy and the original data source would no longer be synchronized.

### 2.3.3 Relations and Mappings

After defining one or more taxonomies a user will most likely want to do something with them. A useful way of working with taxonomies is to create horizontal (as opposed to vertical or parent-child) relations between the nodes in taxonomies or within the same taxonomy. The semantic meaning of these relations can vary, and the user should be able to decide the meaning of these relations.

It would be useful to be able to describe relations as more than just a connection between two nodes. There might be different types of relations between the same nodes and more than one node might participate in a relationship.

When building a large, complex taxonomy it would be useful to have restrictions as to what types of nodes can participate in a relationship, to ensure conformity.

### 2.3.4 Accessibility and compatibility

When a taxonomy and its relations are defined it should be made accessible to the persons or programs that are supposed to use it. Accessibility poses two problems: where and how should data be accessed.

This problem basically has three possible solutions in this context: locally, on a LAN or on the Internet. Since we envision many people working together on the same projects with this framework, there should be at least LAN access (such as within a company) to a taxonomy, but one could quite easily imagine cases where global (Internet) access would be required, such as people from two different companies working together on a project.

This furthermore poses the problem of security. It should be possible to define who can do what. The granularity of access control should be on the node- and relation-levels.

Besides being accessible, taxonomies should also be compatible with common standards. This could be achieved by defining export functionality, so that taxonomies may be exported in various formats such as XML, RDF/OWL etc. In the case of exports, consumers will not be able to edit the taxonomy though, since the exported instance will be decoupled from the original.

## 2.4 Requirements

Based on the previous sections we will now define a number of requirements, that we will have in mind when designing the framework.

### Separation of structural and non-structural data

We want to be able to build taxonomies of any kinds of objects, and it should be possible to define new objects as well. To attain this kind of flexibility it will be necessary to separate structural and non-structural data. This kind of structure also allows us to generate taxonomies from already existing data while avoiding duplicating these data.

The framework must be able to reference non-structural data from various data sources, such as databases, XML documents and web services. But referencing external data sources should not be mandatory.

In some cases it will also be useful to have data that is particular to a taxonomy and applies to nodes of different types. Therefore it must also be possible to define properties on nodes. These properties should be stored with the taxonomy.

### **Rich type system**

More advanced restrictions on vertical (child - parent) as well as horizontal (link) relationships will make the taxonomy framework easier to use, because it becomes easier to avoid errors, in the same way that it is easier to write a computer program when syntax errors are discovered at compile time.

For example, it will be possible to restrict what nodes a specific node can have as children and what links that node can be connected to. Similarly, it will be possible to restrict what nodes a link could be connected to.

### **Persistence**

This might seem like a trivial requirement, but without an easy way of storing taxonomies, they will not prove very useful. It must be possible to store taxonomies in various formats as mentioned above.

The persistence requirement implies the need for a data layer in the framework, and the design goal of various storage formats implies the need for a data access layer, since a consumer should not need to worry about the actual implementation of the data store. With a data access layer, a consumer will not have to worry about storage formats when accessing a taxonomy through the API.

### **Interface**

There must be a standard programming interface (API) for working with taxonomies. All functionality must be defined in this API, and taxonomies should exclusively be accessed through the API.

Included in the API will be a data access layer (DAL) representing a uniform method of accessing taxonomies stored in different formats.

With a standard API users will also be able to define their own transformations. This is important, since it is unlikely that we are able to foresee all transformations that users might need.

### **Transformations**

Standard transformations must be defined according to what is considered useful. Examples will include appending a node to another node and deleting a node.

### **Relations**

It must be possible to create relations to nodes in different taxonomies as well as within the same taxonomy.

Relations between nodes must be typed, and users will be able to define their own relation types.

It must be possible to define what types of nodes can participate in a relation of a specific type and it must be possible to involve more than just two nodes in a relation.

### **Interoperability**

It must be possible to create mappings between nodes that are deemed to have the same or similar semantic value but are placed in different taxonomies. If relations are typed, this can be handled by simply defining a mapping relation and applying that when mapping nodes.

### **Accessibility**

It should be possible to publish a taxonomy on a LAN or on the Internet. This can be achieved by writing a standard web service with methods similar to the methods exposed through the API. When deploying this web service the user will simply specify the data source of the taxonomy and it will be accessible on the Internet. Another way of achieving this is to write a web-based GUI for manipulating taxonomies.

### **Security**

If taxonomies are to be accessible, we will also need to define a security mechanism so that users can limit access to their taxonomies.

The granularity of user rights should be on the node and relation level, meaning that it should be possible to grant or revoke user rights on individual nodes or relations.

This security mechanism must also define what kinds of rights a user can have on nodes and relations, e.g. read, write, create etc.

### **Compatibility**

To ensure compatibility with common standards, such as RDF/OWL, it must be possible to import from and export to these standards. These capabilities should be defined in the API.

Exported taxonomies will be decoupled from the original, and therefore changes will not push through to the original.

## **2.5 Summary**

In this chapter we have described the current Taxonomy Tool and some of its use cases. We have then defined a number of general design goals, and based on these we have defined a number of requirements for the framework.

In the next chapter we will present a brief introduction to bigraphs, including the definition of a formal model for bigraphs.

## Chapter 3

# Bigraphs

In this chapter we will provide a brief introduction to bigraphs, explaining the major concepts and defining the terms used in the rest of the report. We will then present a more formal definition of bigraphs, including a formalized way of defining operations on bigraphs (reaction rules). Finally we will introduce bigraph building blocks, which are a special kind of bigraphs with which all bigraphs can be built.

### 3.1 A short introduction to bigraphs

We will now provide a brief introduction to bigraphs including the background of the theory and the main concepts.

#### 3.1.1 Background

Bigraphs is a mathematically founded data structure developed for modeling process calculi and mobile systems. Most of the research on bigraphs has been carried out by professor Robin Milner of Cambridge University.

Milner has also suggested using bigraphs as a model of ubiquitous computing, one of the 7 grand challenges defined by the UK Computing Research Committee<sup>1</sup>. The connection between ubiquitous computing and bigraphs is explored in [6].

Bigraphs are based on a well founded mathematical model. There exist several formal definitions of bigraphs but they all share the same principles. It can be said that bigraph theory is not yet fully established in a broad sense, and the theory is therefore subject to change. Bigraph theory is originally founded in category theory.

#### 3.1.2 Concepts

A *bigraph* consists of a *place graph* and a *link graph*.

A place graph contains a set of *regions*, and each region contains a forest of *nodes* that may be connected as trees, meaning each node can have either no parent or one parent, and there can be no circular relationships. A region

---

<sup>1</sup><http://www-dse.doc.ic.ac.uk/Projects/UbiNet/GC/index.html>

can be thought of as the root node of the trees that it contains, in which case a bigraph can be thought of as containing a forest (a number of disconnected trees). Some bigraph definitions do not employ the term 'region', but we have chosen to do so in the context of this thesis.

Each node has a *control* and each control has an *arity* which is a number. The arity of a control decides how many *ports* that control has. As we shall see later, controls can be thought of as type definitions for nodes.

Regions may also contain *holes*. Holes can be positioned anywhere a node can be positioned (i.e. they may have a parent), but a hole cannot act as a parent to other nodes or holes. Also, a hole does not have a control, and thus cannot be linked.

The *inner width* of a bigraph is defined as the number of holes in the bigraph and the *outer width* of a bigraph is defined as the number of regions in the bigraph.

A link graph contains a number of *edges* and a bigraph furthermore has a set of *inner names* and *outer names*. Ports and inner names constitute the *points* of a bigraph and edges and outer names constitute the *links* of a bigraph. Within a bigraph any point can be connected to any link.

Nodes are connected to links through their ports. The number of ports a node has and therefore also the number of links a node can be connected to is equal to the arity of the control of the node. Inner names can be connected to an unlimited amount of links.

Two bigraphs can be composed with each other. There are two basic kinds of composition: *vertical composition* and *horizontal composition*.

When two bigraphs,  $A$  and  $B$ , are vertically composed, the one bigraph ( $B$ ) is inserted into the other bigraph ( $A$ ). This can only be done if the outer width and outer names of  $B$  are the same as the inner width and inner names of  $A$ . Composition is then performed by inserting the regions of  $B$  into the holes of  $A$ , and then connecting the edges connected to the outer names of  $B$  with the corresponding edges connected to inner names of  $A$ .

There are two variations of horizontal composition: tensor horizontal composition and parallel horizontal composition.

Horizontal composition can be performed when two bigraphs have disjoint inner names and disjoint outer names. It is done by simply positioning the regions of the one bigraph next to the regions of the other bigraph and then combining their link graphs.

Parallel horizontal composition is similar to normal horizontal composition, except that links connected to the same names in the outer face are merged. This means that the outer name sets need not be disjoint.

A *reaction rule* is like a function that takes a bigraph as its input and then modifies that bigraph in a certain way. Each reaction rule specifies how part of the input bigraph must be structured in order for the reaction rule to be applied to that bigraph.

Bigraphs are usually visualized as topographical graphs, where child nodes are contained within parent nodes. However, because we want to use bigraphs as the data model of an application that handles tree structures, we will visualize them as forests and trees of nodes are usually visualized - with the root at the top and children below parents and nodes connected by lines.

### 3.1.3 Bigraph example: workflow

We shall now give an example of a bigraph. The example is illustrated in figure 3.1.

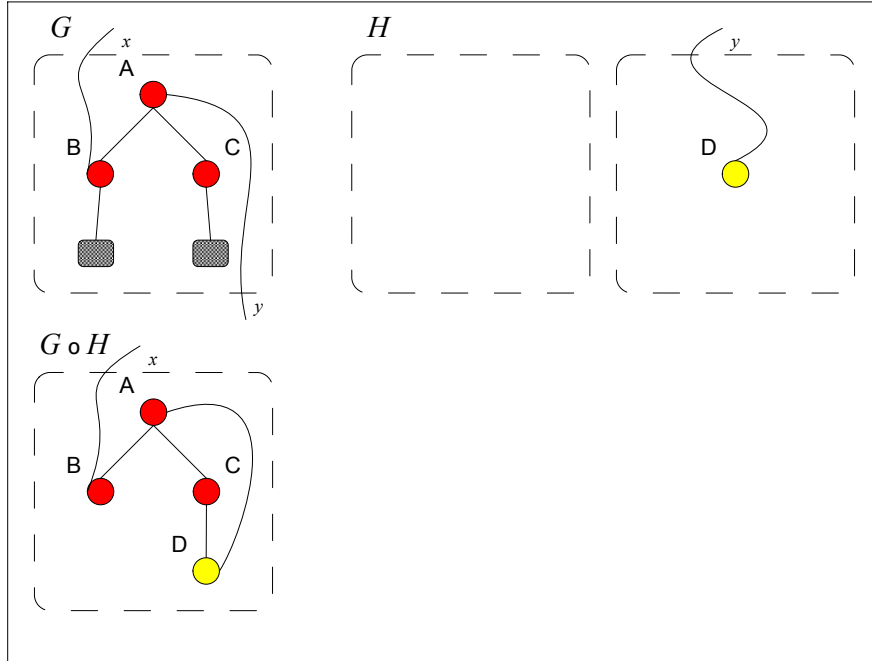


Figure 3.1: An example showing the composition of two bigraphs

The bigraph  $G$  consists of one region. The region contains a small tree of three nodes, with one node ( $A$ ) being the parent of the other two nodes ( $B$  and  $C$ ). Each of the children nodes has a hole as its sole child.

$G$  has one name in the outer face ( $x$ ) and one name in the inner face ( $y$ ). The node  $A$  is connected to the name  $y$  in the inner face and the node  $B$  is connected to the name  $x$  in the outer face.

We can see that  $G$  has an outer name set that consists of the name  $x$  and an outer width = 1. Similarly,  $G$  has an inner name set that consists of the name  $y$  and an inner width = 2.

In order to compose a bigraph  $H$  into  $G$ ,  $H$ 's outer name set and outer width must match  $G$ 's inner name set and inner width.

$H$  consists of 2 regions, but only one of the regions contains any nodes. There is also a name in the outer name set to which the node is connected.

The result of horizontally composing the bigraphs  $G$  and  $H$  is written  $G \circ H$ . The hole that was the child of  $B$  before has now been replaced with the empty region from  $H$  and is therefore gone. The hole that was the child of  $C$  has been replaced with the other region of  $H$ , and thus  $C$  has acquired a child node. The nodes  $A$  and  $D$  are now connected to the same edge, because they were previously connected to edges that were connected to matching names in respectively  $G$ 's inner name set and  $H$ 's outer name set.

This example provides an overview of the basics of bigraphs. We will now proceed to provide a more detailed definition of the theory.

## 3.2 Bigraphs defined

In this section we will introduce a formal definition of bigraphs and reaction rules. Each definition will be followed by a less formal explanation to help the reader interpreting the definition correctly.

After presenting the definitions we will describe how all bigraphs can be built by a set of basic bigraphs (called building blocks).

### 3.2.1 Definitions

Definitions 3.2.1 to 3.2.9 were originally introduced in [5] and definition 3.2.10 was originally introduced in [4]. If one wishes to know more about the mathematical foundations of bigraphs (including bigraphs in the context of category theory), these are good places to start.

Some of the definitions have been shortened to improve readability. Only parts that were deemed not to be necessary in the context of this thesis have been left out.

#### Notation

Bigraph theory is based on category theory, in which the following notation is used: ‘ $\circ$ ’ is used for composition, ‘ $\text{id}$ ’ is used for identity and ‘ $\otimes$ ’ is used for tensor product. In this thesis we will attempt to avoid discussing bigraphs in the context of category theory as much as possible, since we are more interested in bigraphs as data structures. The above-mentioned notation is, however, necessary when it comes to discussing composition of bigraphs.

Furthermore,  $\text{id}_S$  is used to denote the identity function on a set  $S$ .  $S \uplus T$  is used for union of sets known or assumed to be disjoint, and similarly  $f \uplus g$  is used for union of functions with domains known or assumed to be disjoint.

#### Term definitions

A node is *barren* if it has no children. Two nodes are *siblings* if they have the same parent.

A link is *idle* if it is not connected to any node. An outer link is an *open* link, an edge is a *closed* link. A *point* (i.e. an inner name or a port) is *open* if its link is open, otherwise *closed*. Two distinct points are *peers* if they are in the same link. A link graph is *lean* if it has no idle edges.

A bigraph is *prime* if it has only one region. A bigraph is *ground* if it has no holes. A bigraph is *open* if all of its links are connected to the outer face. A bigraph is *discrete* if it is open and no two ports are connected to the same link.

The fully degenerate form  $\epsilon$  is defined as  $\epsilon \stackrel{\text{def}}{=} \langle 0, \emptyset \rangle$ . It is a bigraph with no regions and no names in the outer and inner name name sets.

In the definitions vertical composition is simply referred to as ‘composition’ and horizontal composition is referred to as ‘tensor product’. This is due to bigraph theory’s relation to category theory.

$\mathcal{X}$  is a set of names.

**Definition 3.2.1** (signature). A *signature*  $\mathcal{K}$  is a set whose elements are called *controls*. For each control  $K \in \mathcal{K}$  it provides a finite ordinal  $ar(K)$ , an *arity*.

A signature simply holds the definitions of the controls defined in a specific bigraph. Each control is defined by a name and an arity, the number of links that control can be connected to, and a signature provides a function giving the arity of each control it contains.

**Definition 3.2.2** (interface). An *interface*  $I = \langle m, X \rangle$  consists of a finite ordinal  $m$  called a *width*, a finite set  $X \subset \mathcal{X}$  called a *name set*.

A bigraph has an outer width corresponding to its number of regions and an inner width corresponding to its number of holes. Also, a bigraph has a set of outer names corresponding to the links going out and a set of inner names corresponding to the links going in. The outer face of a bigraph contains its outer width and outer names and the inner face of a bigraph contains its inner width and its inner names.

Interfaces become significant when vertically composing bigraphs. If bigraph  $B$  is to be inserted into bigraph  $A$ , then  $B$ 's outer face must match  $A$ 's inner face.

**Definition 3.2.3** (concrete bigraph). A *concrete bigraph* over the signature  $\mathcal{K}$  takes the form  $G = (V, E, ctrl, G^P, G^L) : I \rightarrow J$  where the interfaces  $I = \langle m, X \rangle$  and  $J = \langle n, Y \rangle$  are its *inner* and *outer faces*. Its first two components  $V$  and  $E$  are finite sets of *nodes* and *edges* respectively. The third component  $ctrl : V \rightarrow \mathcal{K}$ , a *control map*, assigns a control to each node. The remaining two are:

$$\begin{aligned} G^P &= (V, ctrl, prnt) : m \rightarrow n && \text{a place graph} \\ G^L &= (V, E, ctrl, link) : X \rightarrow Y && \text{a link graph.} \end{aligned}$$

This defines the actual bigraph as a whole. The set  $V$  contains all the nodes and the set  $E$  contains all the *edges* of a bigraph. The function  $ctrl$  gives the control from  $\mathcal{K}$  assigned to each node in  $V$ .  $G^P$ , the place graph, and  $G^L$ , the link graph, are defined in 3.2.4 and 3.2.6.

$V$ ,  $E$  and  $ctrl$  are all repeated in the definitions of the place graph and link graph.

$I$  and  $J$ , respectively the inner face and the outer face of the bigraph, are the signatures that define what other bigraphs a bigraph can be vertically composed with, as explained above.

**Definition 3.2.4** (place graph). A *place graph*  $G = (V, ctrl, prnt) : m \rightarrow n$  has an *inner width*  $m$  and an *outer width*  $n$ , both finite ordinals; a finite set  $V$  of nodes with a control map  $ctrl : V \rightarrow \mathcal{K}$ ; and a *parent map*  $prnt : m \uplus V \rightarrow V \uplus n$ . The parent map is *acyclic*, i.e.  $prnt^k(v) \neq v$  for all  $k > 0$  and  $v \in V$ .

A place graph contains a set of nodes ( $V$ ), a function  $ctrl$  that gives the control from  $\mathcal{K}$  assigned to each node in  $V$  and a function  $prnt$  that gives the parent of each node in  $V$ . Furthermore, a place graph has an inner and an outer width, which corresponds to respectively the number of holes and the number of regions in the place graph (see definition 3.2.2).

That the parent map is acyclic means that the trees defined by the place graph are acyclic, i.e. a node cannot be its own ancestor.

**Definition 3.2.5** (composition of place graphs). The sites and roots provide the means of composing the forests of two place graphs; each root  $i$  of the first is planted in site  $i$  of the second. Formally, let  $G_i = (V_i, ctrl_i, prnt_i) : m_i \rightarrow m_{i+1}$  ( $i = 0, 1$ ) be place graphs with  $V_0 \cap V_1 = \emptyset$ . Then  $G_1 \circ G_0 \stackrel{def}{=} (V, ctrl, prnt)$  where  $V = V_0 \uplus V_1$ ,  $ctrl = ctrl_0 \uplus ctrl_1$ , and

$$prnt = (\text{ld}_{V_0} \uplus prnt_1) \circ (prnt_0 \uplus \text{ld}_{V_1}).$$

The identity place graph at  $m$  is  $\text{id}_m \stackrel{def}{=} (\emptyset, \emptyset_{\mathcal{K}}, \text{ld}_m) : m \rightarrow m$ .

The tensor product of two place graphs  $G : k \rightarrow \ell$  and  $H : m \rightarrow n$  with disjoint node sets is  $G \otimes H : k + m \rightarrow \ell + n$ . It consists simply of placing the two forests side-by-side, and we need not define it more formally.

Only bigraphs with disjoint sets of nodes can be vertically composed. Vertical composition of one bigraph ( $B$ ) into another bigraph ( $A$ ) is carried out by joining the two sets of nodes and the two sets of node-to-control mappings, and then combining the two parent mappings, such that each root node in  $B$  becomes the child of a node in  $A$ . This is done by matching each hole in  $A$  with a region from  $B$ , and then replacing that hole with the region. Matching is done by simple ordering.

It is assumed that the set of node names of the parent mapping  $A$  is disjoint with the set of node names in  $B$ . Since the number of regions in  $B$  and the number of holes in  $A$  must match, it is also assumed that the inner face of  $A$  matches the outer face of  $B$ .

The identity place graph  $\text{id}_m$  simply denotes the bigraph with  $m$  regions, each containing one hole.

Tensor horizontal composition of two bigraphs also requires that their node sets be disjoint, but there are no requirements for the interfaces. Horizontal composition is carried out by simply inserting the regions of  $B$  into  $A$  so that they are positioned next to the original regions of  $A$ .

**Definition 3.2.6** (link graph). A *link graph*  $G = (V, E, ctrl, link) : X \rightarrow Y$  has finite sets  $X$  of *inner names*,  $Y$  of *outer names*,  $V$  of *nodes* and  $E$  of *edges*. It also has a function  $ctrl : V \rightarrow \mathcal{K}$  called the *control map*, and a function  $link : X \uplus P \rightarrow E \uplus Y$  called the *link map*, where the disjoint sum  $P \stackrel{def}{=} \sum_{v \in V} ar(ctrl(v))$  is the set of *ports* of  $G$ .

A link graph contains a set of nodes ( $V$ ) and a set of edges ( $E$ ), a function  $ctrl$  that gives the control from  $\mathcal{K}$  assigned to each node in  $V$  and a function  $link$  that gives the edge from  $E$  or outer name that each port on each node in  $V$  and each inner name is connected to.

**Definition 3.2.7** (composition of link graphs). [...] Formally, let  $G_i = (V_i, E_i, ctrl_i, link_i) : X_i \rightarrow X_{i+1}$  ( $i = 0, 1$ ) be two link graphs with  $V_0 \cap V_1 = E_0 \cap E_1 = \emptyset$ . Then  $G_1 \circ G_0 \stackrel{def}{=} (V, E, ctrl, link)$  where  $V = V_0 \uplus V_1$ ,  $ctrl = ctrl_0 \uplus ctrl_1$ ,  $E = E_0 \uplus E_1$  and

$$link = (\text{ld}_{E_0} \uplus link_1) \circ (link_0 \uplus \text{ld}_{P_1}).$$

We can describe the composite link map  $link$  of  $G_1 \circ G_0$  as follows, considering

all possible arguments  $p \in X_0 \uplus P_0 \uplus P_1$ :

$$\text{link}(p) = \begin{cases} \text{link}_0(p) & \text{if } p \in X_0 \uplus P_0 \text{ and } \text{link}_0(p) \in E_0 \\ \text{link}_1(x) & \text{if } p \in X_0 \uplus P_0 \text{ and } \text{link}_0(p) = x \in X_1 \\ \text{link}_1(p) & \text{if } p \in P_1. \end{cases}$$

The identity link graph at  $X$  is  $\text{id}_X \stackrel{\text{def}}{=} (\emptyset, \emptyset, \emptyset_{\mathcal{K}}, \text{id}_X) : X \rightarrow X$ .

The tensor product of two link graphs  $G : W \rightarrow X$  and  $H : Y \rightarrow Z$  can be formed provided that their node sets and edge sets are disjoint and that  $W \cap Y = X \cap Z = \emptyset$ . It is  $G \otimes H : W \uplus Y \rightarrow X \uplus Z$ , and consists simply of the union of their link maps.

When performing vertical composition, the two link graphs (again we use the example of composing  $B$  into  $A$ ) are assumed to have disjoint sets of nodes and edges, and therefore the node and edge sets and the *ctrl* function are simply combined. The new *link* function of  $A$  then becomes the result of composing the two *link* functions, such that all mappings for points in  $B$  that are not connected to the outer face are unchanged, all mappings for points in  $B$  that are connected to a name in the outer face of  $B$  are connected to the corresponding link in the inner face of  $A$  and all mappings for points in  $A$  are unchanged.

The identity link graph  $\text{id}_X$  is simply a bigraph with no regions and the set of names  $X$  in both the outer and inner face.

Tensor horizontal composition of two link graphs that have disjoint node and edge sets, and disjoint inner names and outer names is simply the union of their link maps.

**Definition 3.2.8** (composition of concrete bigraphs). The composition of two concrete bigraphs  $G = \langle G^P, G^L \rangle : I \rightarrow J$  and  $H = \langle H^P, H^L \rangle : J \rightarrow K$  with disjoint node sets and disjoint edge sets is

$$H \circ G \stackrel{\text{def}}{=} \langle H^P \circ G^P, H^L \circ G^L \rangle : I \rightarrow K.$$

The vertical composition of two concrete bigraphs is defined as the vertical composition of their place graphs and the vertical composition of their link graphs. If  $B$  is vertically composed into  $A$  the resulting bigraph will then have the inner names of  $B$  and the outer names of  $A$ .

But this definition is slightly prohibitive, in that it only allows us to vertically compose bigraphs that have disjoint node and edge sets. In category theory terms, this is only a precategory, not a category. Therefore abstract bigraphs are introduced.

**Definition 3.2.9** (composition of abstract bigraphs). Two concrete bigraphs  $G_0$  and  $G_1$  are said to be *lean-support equivalent*,  $G_0 \simeq G_1$ , if they differ only by a bijection between their nodes and their non-idle edges; idle edges are ignored. An *abstract bigraph* consists of a  $\simeq$ -equivalence class of concrete bigraphs. Compositions and identity of abstract bigraphs are given by

$$\begin{aligned} [H]_{\simeq} \circ [G]_{\simeq} &\stackrel{\text{def}}{=} [ \langle H^P \circ G^P, H^L \circ G^L \rangle ]_{\simeq} \\ \text{id}_{\langle m, X \rangle} &\stackrel{\text{def}}{=} [ \langle \text{id}_m, \text{id}_X \rangle ]_{\simeq}. \end{aligned}$$

The tensor of two interfaces with disjoint name sets is

$$\langle m, X \rangle \otimes \langle n, Y \rangle \stackrel{def}{=} \langle m + n, X \uplus Y \rangle.$$

The tensor product of two abstract bigraphs  $F : H \rightarrow I$  and  $G : J \rightarrow K$ , where  $H \otimes J$  and  $I \otimes K$  are defined, is given by

$$[F]_{\varnothing} \otimes [G]_{\varnothing} \stackrel{def}{=} [\langle F^P \otimes G^P, F^L \otimes G^L \rangle]_{\varnothing} : H \otimes J \rightarrow I \otimes K.$$

When testing for equivalence of abstract bigraphs, idle edges and id of nodes and edges are ignored. Therefore we can now define composition of bigraphs with non-disjoint sets of node and edge id's. The definition is very similar to the definition of composition of concrete bigraphs, but the signatures are left out.

As with concrete bigraphs, horizontal composition of two abstract bigraphs is defined as the horizontal composition of their place graphs, the horizontal composition of their link graphs and the horizontal composition of their signatures.

Abstract bigraphs should not be conceived so much as a different type of bigraphs, but rather as a special view of concrete bigraphs where certain details are disregarded. Therefore all concrete bigraphs can also be represented as abstract bigraphs. Practically, vertical composition of abstract bigraphs with non-disjoint node and edge id's is usually carried out by changing the conflicting id's.

**Definition 3.2.10** (reaction rules for bigraphs). A *ground (reaction) rule* is a pair  $(r, r')$ , where  $r$  and  $r'$  are ground with the same outer face. Given a set of ground rules, the *reaction relation*  $\longrightarrow$  over agents is the least, closed under support equivalence ( $\simeq$ ), such that  $D \circ r \longrightarrow D \circ r'$  for each active  $D$  and each ground rule  $(r, r')$ .

A *parametric (reaction) rule* has a *redex*  $R$  and *reactum*  $R'$ , and takes the form

$$(R : I \rightarrow J, R' : I' \rightarrow J, \varrho)$$

where the inner faces  $I$  and  $I'$  are local with widths  $m$  and  $m'$ . The third component  $\varrho :: I \rightarrow I'$  is an instantiation. For every  $X$  and discrete  $d : X \otimes I$  the parametric rule generates the ground reaction rule

$$((id_X \otimes R) \circ d, (id_X \otimes R') \circ (\varrho(d))).$$

*Support equivalence* is similar to lean-support equivalence except that idle edges are still considered when testing for equivalence.

Applying a reaction rule to a bigraph is done by first matching the bigraph ( $A$ ) to the redex of the reaction rule ( $r$  or  $R$ ). A bigraph matches a redex when the whole or a part of the bigraph has the same structure as the whole of the redex. If the redex contains holes these are interpreted as 'wildcards', that is any subtree can be substituted for a hole during a match. The part of  $A$  that matches  $r$  or  $R$  is then transformed so that it matches  $r'$  or  $R'$ .

When searching for matches it is quite possible that there will be more than one, which means that applying a reaction rule to a bigraph may have several possible outcomes. There are no rules defined as to how matches are prioritized.

A ground reaction rule is defined by a tuple of ground bigraphs. To apply the reaction rule to some bigraph  $A$ ,  $A$  must contain a part that is support equivalent to  $r$ . Then  $A = D \circ r$ , where  $D$  is the part of  $A$  that does not match any part of  $r$  (also called the *context*). The result of applying the reaction rule is then  $D \circ r'$ , that is, the original bigraph  $A$  where the part that matches  $r$  is transformed so that it now matches  $r'$ .

A parametric reaction rule works similarly, but the definition also includes a parameter mapping  $\varrho$  (called an instantiation). A parametric reaction rule is defined by a tuple of bigraphs  $((R, R'))$  that may contain holes. The bigraph to which the parametric reaction rule is applied is  $A = D \circ (id_X \otimes R) \circ d$  and the result of applying the reaction rule is then  $A' = D \circ (id_X \otimes R') \circ (\varrho(d))$ .

The parameter map  $\varrho$  is a mapping from one bigraph  $(E)$  to another  $(E')$ . Each hole in  $E'$  is mapped to a hole in  $E$ . This means that some holes in  $E$  may not be mapped to any hole in  $E'$  and some holes in  $E$  may be mapped to more than one hole in  $E'$ .

$d$  denotes the parts of  $A$  that will be inserted into the holes of  $R'$ , and in this context can be thought of as a bigraph on its own. All regions in  $d$  must be discrete.  $X$  denotes the set of outer names in  $d$  and  $id_X$  is included in the definition to preserve these names. When the reaction rule is applied, each region in  $d$  is mapped to zero or more of the holes in  $R'$  according to  $\varrho$ .

### 3.3 Bigraph building blocks

There are two basic types of bigraph building blocks: *placings* and *wirings*. Additionally, there is a building block called a *discrete ion* which fits in neither category.

It is possible to build all imaginable bigraphs with these six building blocks combined with vertical and horizontal composition. Readers who wish to see the mathematical proof of this should refer to [5]. We will simply give an example of how this can be done, after presenting the building blocks.

#### 3.3.1 Placings

Placings are bigraphs with empty link graphs. Thus, they are used for building the place graphs of bigraphs.

There are three kinds of placings:

$1 : \epsilon \rightarrow 1$	a barren region
$merge : 2 \rightarrow 1$	map two sites to one root
$\gamma_{m,n} : m + n \rightarrow n + m$	swap $m$ with $n$ places

The placing  $1$  is simply a bigraph with one barren region.

The placing  $merge$  merges two regions. For all  $m > 0$  there is also defined  $merge_m : m \rightarrow 1$ . That is, a  $merge$  that merges  $m$  regions.  $merge_0$  is defined as the placing  $1$ .

$merge_1$  is equal to the identity bigraph  $id_1$  which is a bigraph with one region which contains one hole.

The placing  $\gamma_{m,n}$  reorganizes the order of the regions within a bigraph, so that the last  $n$  regions become the first  $n$  regions. The order of the regions within those two ranges that are reorganized remains unchanged.

### 3.3.2 Wirings

Wirings are bigraphs with empty place graphs. Thus, they are used for building the link graphs of bigraphs.

There are two kinds of wirings:

$$\begin{array}{ll} /x : x \rightarrow \epsilon & \text{closure} \\ y/X : X \rightarrow y & \text{substitution } x \mapsto y \text{ (all } x \in X) \end{array}$$

A closure simply closes a link so that it does not extend to the outer face.

A substitution connects all the names in the inner face to one name in the outer face. Note, that substitutions are also used to introduce edges. This is done with a substitution that has only one name in the inner face.

### 3.3.3 Discrete ions

A discrete ion is defined as  $K_{\vec{x}} : 1 \rightarrow \langle 1, \vec{x} \rangle$ , for any sequence  $\vec{x} = x_1, \dots, x_k$  of distinct names where  $k = ar(K)$ .

A discrete ion is a discrete, prime bigraph. We recall that discrete means that the bigraph is open (all edges are connected to the outer face) and no two ports are connected to the same point, and prime means that it has just one region. A discrete ion contains one node with a hole as its only child.

### 3.3.4 Building blocks example

We will now present an illustrated example of how a bigraph can be built with building blocks and vertical and horizontal composition. The illustration of each step can be found from page 27 and onwards.

In this example all the bigraph nodes will share the control  $a$ .  $a$  has  $ar(a) = 1$ , meaning that all discrete ions created with this control will have one name in the outer face.

In the diagrams the color of a node merely signifies what bigraph that node originates from.

1. We begin by defining the bigraph  $G$  as the result of composing a discrete ion with a *merge*:

$$G = a_x \circ \text{merge}$$

2. We then define a bigraph  $H$  consisting of one region containing one node:

$$H = (/x \otimes \text{id}_1) \circ (a_x \circ 1)$$

3. We then define a bigraph  $I$  consisting of one region. The region contains a node with a hole as its sole child, and a name in the inner face to which the node is connected:

$$I = ((/x \circ x/\{x, y\}) \otimes \text{id}_1) \circ (a_x \otimes y/\{x\})$$

4.  $H$  and  $I$  are then horizontally composed into each of the holes in  $G$ :

$$G \circ (H \otimes I)$$

5. We then add a second region containing a hole to  $G$ :

$$G \otimes \text{id}_1$$

6. We then define a bigraph  $J$  which has two regions, one of them containing a node the other containing a hole. The node is linked to a name in the outer face:

$$J = (a_x \circ 1) \otimes \text{id}_1$$

7. We then horizontally compose  $J$  into  $G$ . Before we do this, however, we swap the regions of  $J$ :

$$G \circ (\gamma_{1,1} \circ J)$$

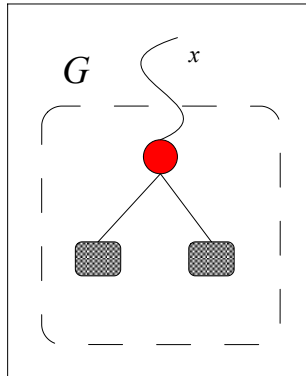
### 3.4 Summary

In this chapter we have presented a formal definition of bigraphs, including a definition of reaction rules. We have also presented the six building blocks from which all bigraphs can be constructed.

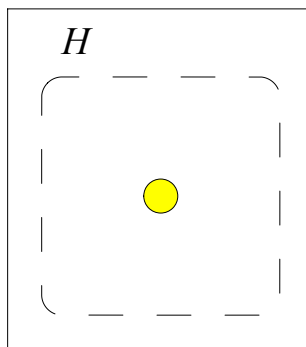
We will now move on to an analysis of the problems involved when designing the taxonomy framework. This analysis will be divided into two chapters. The first chapter will concentrate on the data structure, and the second chapter will concentrate on the architecture of the framework.

**Building block example steps**

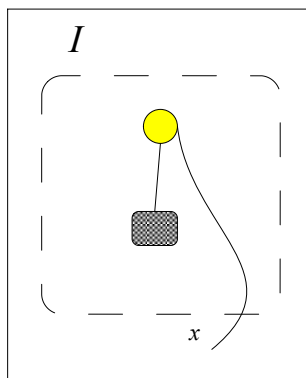
Step 1



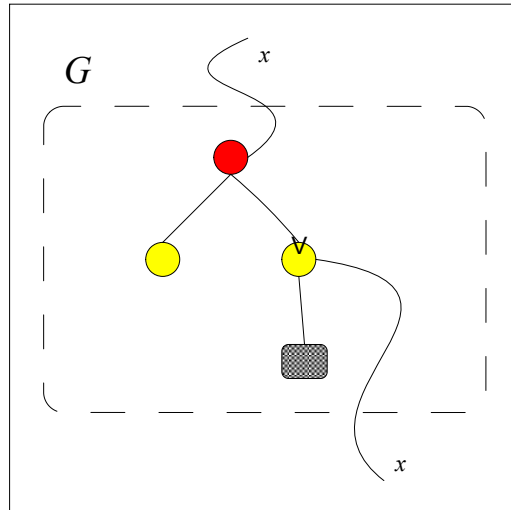
Step 2



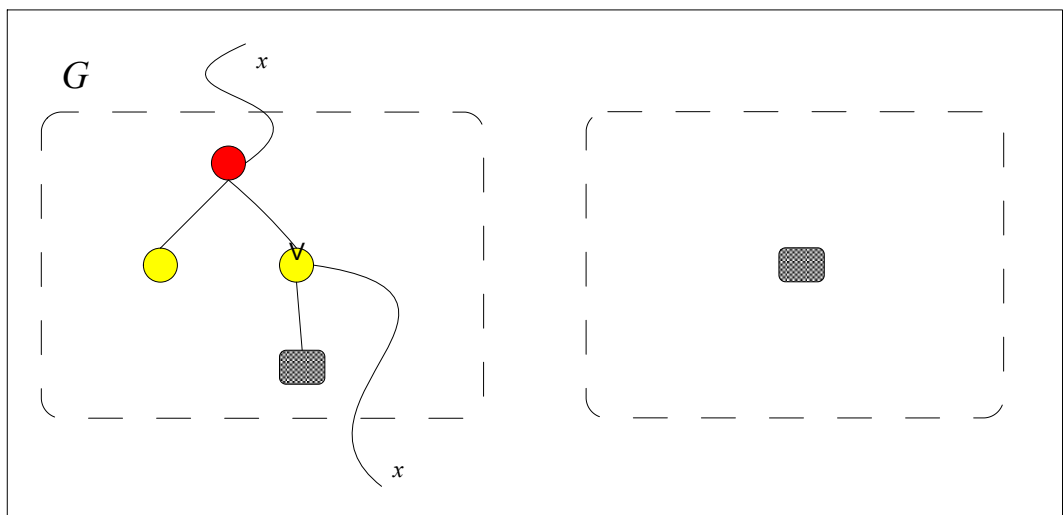
Step 3



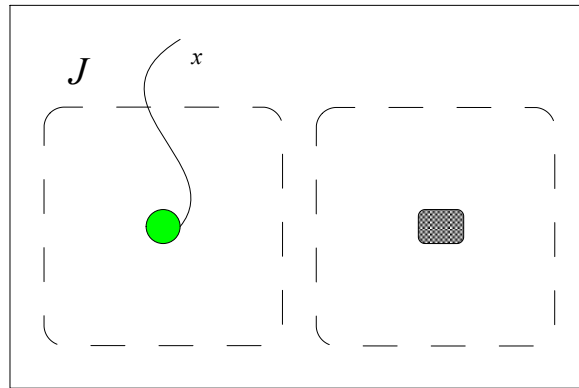
Step 4



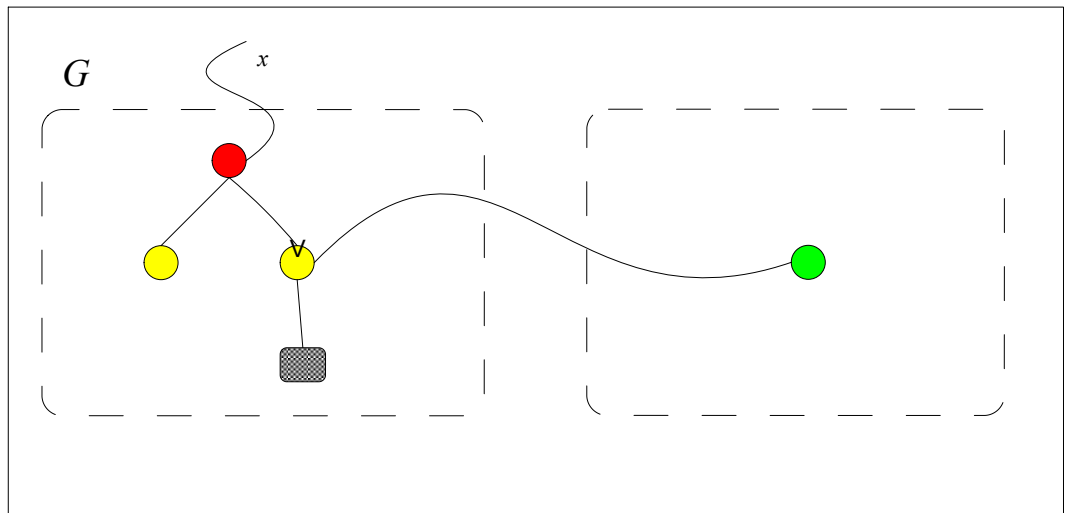
Step 5



Step 6



Step 7



## Chapter 4

# Data Structure Analysis

The internal data structure of the framework will be based on bigraphs. In this chapter we will provide a comparative analysis involving bigraphs. We will focus on identifying the advantages and disadvantages of employing bigraphs in this context by suggesting how the framework could be implemented with other data structures. We will discuss advantages and disadvantages of each data structure and based on this we will discuss what extensions need to be defined to the bigraph theory in order for the framework to satisfy the requirements defined in chapter 2.

### 4.1 Analysis background

When choosing the data structure of the framework we are faced with several options.

We could make a new data structure completely from scratch. This is the most flexible option as we would have complete freedom as to the design and features of the data structure.

We could also choose to implement a data structure based on a formal model. This is what we are doing with bigraphs in this thesis. A formal model provides us with the advantages of a data structure which has been proven to function in theory and a model description that is independent from the implementation. An independent model description ensures that there can be no confusion as to how the data structure functions and it is relatively easy to convey this information. However, as is the case with bigraphs, a formal model might not satisfy all our requirements and we might therefore have to extend the model.

Finally, we could choose a data structure that already has been implemented and therefore has a functional API. Such a data structure would (presumably) have the advantages of wide recognition and an already existing and therefore well tested API. Furthermore, if the data structure is widely recognized it would probably also have a model description that is independent from the implementation. As with the formal model option above an existing implementation might not satisfy all our requirements and in such a case extensions would have to be provided.

We will now present a short survey of some of the data structures the framework could be based on. We will discuss how suitable each data structure would

be as the internal data structure of the taxonomy framework, and finally we will compare the different data structures.

The most basic property of the framework is that it is based on taxonomies, i.e. trees of objects. Therefore, the internal data structure must somehow be able to handle this. Also, the data structure should be able to handle relations between objects in the trees.

Some of the requirements we do not really expect to find inherent in any data structures. For example, separation of structural and non-structural data and external links in the nodes is rather specific to the taxonomy framework. Such features will probably have to be implemented as extensions to whatever data structure is chosen.

## 4.2 XML

XML (eXtensible Markup Language) is not really a data structure but rather a data representation format with which data structures can be defined.

It is interesting in this context, however, because XML is hierarchically structured, which makes it suitable for representing taxonomies which are also hierarchically structured. Therefore, were we to design a completely new data structure ourselves, XML might be a good choice for data representation.

XML has one or more well defined APIs for most programming languages. The .NET class library alone has APIs based on the Document Object Model (DOM) and the Simple API for XML (SAX). Therefore, a simple solution could be to define an XML schema for representing bigraphs in XML and basing the implementation on this. A large part of the logic would be provided through the existing APIs, although additional logic would have to be implemented for e.g. composition and reaction rules.

The Java-based API Java Architecture for XML Binding (JAXB)<sup>1</sup> takes this idea one step further. With JAXB it is possible to generate an object model from an XSD schema file. If we chose to do this the interface would not be a generic XML interface, but an interface specific to the bigraph data format that we would have defined. It would still, however, be necessary to provide the additional logic as described above and we would have to implement the framework in Java.

XML furthermore offers several advantages on account of its wide recognition and simple structure. XML provides a low learning barrier for other developers and it can be easily transferred over computer networks. Also, it provides simple solutions for storage, namely as XML document files or in native XML database solutions, although this is still considered an emerging technology by many.

Since XML does not provide any functionality as such we will not describe each use case in this context.

## 4.3 OWL

The Web Ontology Language (OWL) is tightly connected to the Semantic Web project and the Resource Description Framework (RDF) [11].

---

<sup>1</sup><http://java.sun.com/developer/technicalArticles/WebServices/jaxb/>

The purpose of the Semantic Web project is to create a “universal medium for the exchange of data”. Data on the web should be machine-understandable so that machines can share data without human interaction. RDF is a framework for representing objects (or resources) and relations between them. RDF is usually implemented in XML. OWL extends RDF by adding more vocabulary for describing properties and classes [9].

With RDF it is possible to define subclasses as well as subproperties (properties inheriting from other properties). OWL ontologies consist of classes subclassing other classes and individuals (instances of classes). Individuals can be related through properties.

For example, the class Person might have the subclasses Man and Woman and these might also have subclasses. Person might itself be a subclass of the class Mammal. Also, Mads might be an individual of the type Man and Mads might have a Sister property set to the value of Mette, an individual of type Woman, saying that Mette is Mads’ sister (and Mads is Mette’s brother). Thus we see that OWL ontologies have a hierarchical structure and relationships. RDF allows for multiple inheritance, meaning that a class can subclass more than one class.

OWL furthermore extends RDF making it possible to assign a domain to properties so that only certain classes can be assigned that property, and properties can have a range defined limiting what can be assigned as the value of a property. Also, classes and properties can be defined to be equivalent to (or different from) other classes and properties, and pairs of properties can be defined to be inverse, transitive or symmetric.

One limitation of OWL is that it only has binary relationships, i.e. relationships that involve exactly two entities. This could be overcome by defining a relationship as a class and then defining a property on each participant in the relationship.

Because RDF is usually implemented in XML and OWL extends RDF, OWL is also usually implemented in XML. Fig. 4.1 is an example of OWL implemented as XML based on the example above (namespace declarations have been left out). For a full OWL reference, please refer to [8].

First the class Person and the class Woman, which subclasses Person, is declared. Then the property sister is declared. The domain of the sister property has the value Person, meaning that only individuals of the Person class and its subclasses can have this property. It also has a range with a value of Woman meaning that the value of this property must be an individual of the Woman class or one of its subclasses. Then the class Man is declared. Just like the Woman class, the Man class subclasses the Person class. The Man class can furthermore have a sister property although this is not obligatory since it has a minimum cardinality of 0.

Finally we create the Woman individual Mette and the Man individual Mads. Mads has a sister property set to the value of Mette.

### 4.3.1 OWL-based use case implementations

We will now discuss how the use cases from section 2.2 would be handled if the taxonomy framework was implemented with OWL as its base data structure.

```

<owl:Class rdf:ID="Person" />

<owl:Class rdf:ID="Woman">
  <rdfs:subClassOf rdf:resource="#Person"/>
</owl:Class>

<owl:ObjectProperty rdf:ID="sister">
  <rdfs:domain rdf:resource="#Person" />
  <rdfs:range rdf:resource="#Woman" />
</owl:ObjectProperty>

<owl:Class rdf:ID="Man">
  <rdfs:subClassOf rdf:resource="#Person"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#sister"/>
      <owl:minCardinality rdf:datatype="&xsd;nonNegativeInteger">0
    </owl:minCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

<Woman rdf:ID="Mette" />

<Man rdf:ID="Mads" >
  <sister rdf:resource="#Mette" />
</Man>

```

Figure 4.1: An RDF/OWL example

### Hierarchical object structures

The government forms taxonomy is not an ontology, but it would be possible to build an ontology on which to base the taxonomy.

This ontology would include a Ministry class, an Authority class (each authority is governed by one ministry) and a Form class (each form is governed by one authority or, in some cases, ministry). In order to relate a form to e.g. a law, a law property could be defined for the Form class, and each form individual could then have this property set to the value of the law that it is related to. Of course, this would also require the definition of an ontology of laws.

Fig. 4.2 and fig. 4.3 shows an example of a limited implementation of this (ministries are left out). The `TradeReport` form has `formID` 1 and is governed by the `ToIdSkat` authority. It also has a property showing that it is required by the law `SomeTaxSubParagraph`, which is part of the `SomeTaxParagraph` paragraph.

In the full implementation each of the ministries and authorities would be represented by individuals and they would be related through properties. There would be individuals for each of the forms (only one form is included in the example). In order to relate each form to either a ministry or an authority, the Min-

```

<owl:Class rdf:ID="Law" />

<owl:Class rdf:ID="Paragraph">
  <rdfs:subClassOf rdf:resource="#Law"/>
</owl:Class>

<owl:ObjectProperty rdf:ID="paragraph">
  <rdfs:domain rdf:resource="#SubParagraph" />
  <rdfs:range rdf:resource="#Paragraph" />
</owl:ObjectProperty>

<owl:Class rdf:ID="SubParagraph">
  <rdfs:subClassOf rdf:resource="#Law"/>
  <owl:Restriction>
    <owl:onProperty rdf:resource="#paragraph"/>
    <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">1
  </owl:cardinality>
  </owl:Restriction>
</owl:Class>

<Paragraph rdf:ID="SomeTaxParagraph" />

<SubParagraph rdf:ID="SomeTaxSubParagraph" >
  <paragraph rdf:resource="#SomeTaxParagraph" />
</ SubParagraph>

```

Figure 4.2: The laws taxonomy implemented as RDF

istry and Authority classes would have to subclass a `GovernmentInstitution` class. Similarly, a form can be related to zero or more laws through the `requiredBy` property, the value of which is a `Law` (i.e. either a `Paragraph` or a `SubParagraph`).

### Workflows

Again, workflows and ontologies are not really the same, but OWL could also be employed to represent workflows.

Fig. 4.4 shows a simple example of how workflows could be implemented in RDF. Two classes are defined: `Input` and `WorkflowStep`. The details of the `Input` class are not included here, but it represents a generic class for defining the input in a single workflow step. The `WorkflowStep` has an `Input` individual and zero or more branches, which are references to other `WorkflowStep` individuals. Each branch represents a possible next step in the workflow. The logic for determining the next step is left unspecified.

The figure also includes an implementation of the example from section 2.2.3. Since step 3 has no branches, it is determined to be the last branch. The `Input` individuals are left out as they are not important in this example.

```

<owl:Class rdf:ID="Authority" />

<owl:DatatypeProperty rdf:ID="formID">
  <rdfs:domain rdf:resource="#Form" />
  <rdfs:range rdf:resource="&xsd;positiveInteger"/>
</owl:DatatypeProperty>

<owl:ObjectProperty rdf:ID="authority">
  <rdfs:domain rdf:resource="#Form" />
  <rdfs:range rdf:resource="#Authority" />
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="requiredBy">
  <rdfs:domain rdf:resource="#Form" />
  <rdfs:range rdf:resource="#Law" />
</owl:ObjectProperty>

<owl:Class rdf:ID="Form">
<rdfs:subClassOf>
  <owl:Restriction>
    <owl:onProperty rdf:resource="#formID"/>
    <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">1
  </owl:Restriction>
</rdfs:subClassOf>
<rdfs:subClassOf>
  <owl:Restriction>
    <owl:onProperty rdf:resource="#authority"/>
    <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">1
  </owl:Restriction>
</rdfs:subClassOf>
<rdfs:subClassOf>
  <owl:Restriction>
    <owl:onProperty rdf:resource="#requiredBy"/>
    <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">0
  </owl:Restriction>
</rdfs:subClassOf>
</owl:Class>

<Form rdf:ID="TradeReport" >
  <formID rdf:datatype="&xsd;positiveInteger">1</formID>
  <authority rdf:resource="#ToldSkat" />
  <requiredBy rdf:resource="#SomeTaxSubParagraph" />
</Form>

```

Figure 4.3: The forms taxonomy implemented as RDF

```

<owl:Class rdf:ID="Input" />

<owl:ObjectProperty rdf:ID="input">
  <rdfs:domain rdf:resource="#WorkflowStep" />
  <rdfs:range rdf:resource="#Input" />
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="branch">
  <rdfs:domain rdf:resource="#WorkflowStep" />
  <rdfs:range rdf:resource="#WorkflowStep" />
</owl:ObjectProperty>

<owl:Class rdf:ID="WorkflowStep">
<rdfs:subClassOf>
  <owl:Restriction>
    <owl:onProperty rdf:resource="#branch"/>
    <owl:minCardinality rdf:datatype="&xsd;nonNegativeInteger">0
  </owl:minCardinality>
  </owl:Restriction>
</rdfs:subClassOf>
</owl:Class>

<WorkflowStep rdf:ID="step1" >
  <input rdf:ID="CompanyInformation" />
  <branch rdf:ID="step2a" />
  <branch rdf:ID="step2b" />
<\ WorkflowStep>

<WorkflowStep rdf:ID="step2a" >
  <input rdf:ID="DomesticTrade" />
  <branch rdf:ID="step3" />
<\ WorkflowStep>

<WorkflowStep rdf:ID="step2b" >
  <input rdf:ID="InternationalTrade" />
  <branch rdf:ID="step2a" />
<\ WorkflowStep>

<WorkflowStep rdf:ID="step3" >
  <input rdf:ID="Final" />
<\ WorkflowStep>

```

Figure 4.4: A workflow implemented in RDF

### 4.3.2 Summary

The main advantage of OWL is its class and property system. It is possible to build very sophisticated ontologies, and then base collections of objects on these. This equips systems based on OWL with a feature rich type system.

If OWL was used as the internal data structure of the taxonomy framework the type system requirements would certainly be satisfied and we would be able to provide advanced link functionality through properties.

However, OWL has a very specific application domain, namely that of ontologies and, although these are hierarchically structured ontologies, OWL has shortcomings when it comes to representing generic hierarchical structures.

Since the nodes in OWL represent concepts and ideas, the nodes in the OWL-based taxonomies are basically types. The instances of these types can also be part of the taxonomies, but the OWL taxonomies are supposed to be built mainly with types.

In the taxonomy framework we also require type information, but the taxonomies themselves should not be built from this. The taxonomies are supposed to be built from instances of types, i.e. objects.

We can therefore conclude that there are issues with the way the type system works when also taking into consideration the way the hierarchical structures are built.

## 4.4 Bigraphs

Bigraphs were introduced in chapter 3. Bigraphs offer features that are distinctly different from those offered by XML and OWL, e.g. compositionality and reaction rules.

### 4.4.1 Bigraph-based use case implementations

We will now examine how the use cases from section 2.2 could be implemented employing bigraphs and we will then discuss how suited bigraphs are as the underlying data structure of the taxonomy framework.

#### **Hierarchical object structures**

Bigraphs are basically hierarchical object structures with links between them, so therefore they are potentially well suited for this use case.

Figure 4.5 shows a bigraph implementation of the forms and laws taxonomies example from section 2.2.1.

The taxonomies contain authority, form, paragraph (§) and subparagraph (§§) nodes. The forms are children of an authority and the subparagraphs are children of a paragraph. Two of the forms are connected to laws.

However, just like the old Taxonomy Tool, the type system of bigraphs is not very feature rich. The type of a node only restricts the amount of horizontal relations that a node can participate in.

Therefore this kind of use case would benefit even further if the type system was extended as explained in section 2.4.

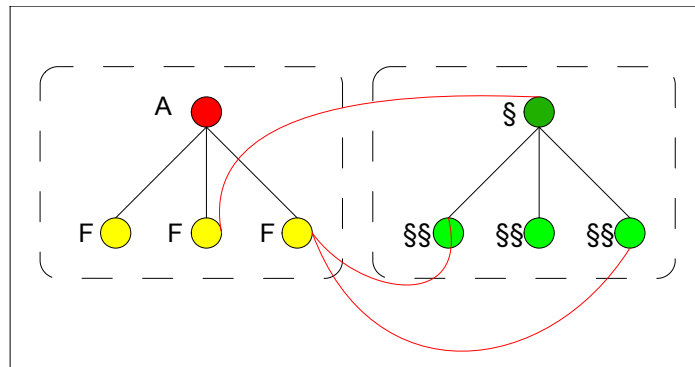


Figure 4.5: The forms and laws taxonomies as bigraphs

**Workflows**

Again, since the structure of bigraphs is inherently hierarchical they are potentially well suited for creating workflows. Bigraphs do not carry properties, but they could easily be extended to do this.

In particular, bigraphs are suitable for creating shareable components because the logic for composing bigraphs is well defined. This could be done by defining each workflow step as a small bigraph, and then composing them together to build the final workflow.

This is illustrated in figure 4.6, which is the example from section 2.2.3 implemented as a bigraph. Only the structure of the workflow is implemented here. Details such as input fields and next step decision logic is left out.

Each workflow step is defined as a small bigraph. The grey boxes inside the first three bigraphs are the holes where another bigraph is placed by composition.

The workflow is created by composing the individual workflow steps together. The last workflow step has no hole in it, so other workflow steps cannot be composed into it. This implies that it is the last step in the workflow.

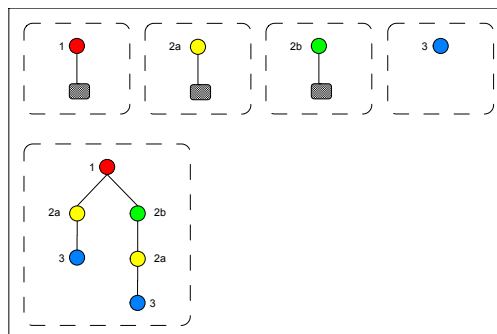


Figure 4.6: A workflow implemented as a bigraph

The persisted bigraph could be used simply as the structure of the workflow, with external logic taking care of persistence of input data, or it could be thought of as a base workflow with a new bigraph instance created whenever a user starts

on a workflow.

Additionally, the data that is input at each workflow step could be thought of as external data, and could then be persisted to different data sources according to the control type of a node. E.g., general company data could be stored in one database and data regarding that specific workflow might be stored in a separate database.

In this case it is clear that using bigraphs as the underlying data structure, opens up for completely new ways of using the taxonomy framework.

As we have seen that bigraphs support sharing of structures we can also conclude that the more general case of workflows mentioned in section 2.2.3 could be implemented with bigraphs. We could imagine each pattern defined as a bigraph. Workflows could then be built by composing these pattern bigraphs and reaction rules might be defined to provide the dynamic logic of the workflow engine.

#### 4.4.2 Summary

The main advantages of bigraphs are compositionality and reaction rules. Composition can be used for building taxonomies out of predefined components that can be shared, thereby simplifying the processes of creation and maintenance and reaction rules provide bigraphs with well defined dynamic logic.

Another important feature of bigraphs is that they are generic data structures with a wide application domain, something that corresponds well with the nature of the taxonomy framework.

Also, the fact that bigraphs are based on a formal model means that bigraph-based applications will have an implementation independent model description as discussed earlier.

Although bigraphs are inherently hierarchical structures, the type system of bigraphs has limited functionality. Therefore it will be necessary to extend this type system. They also do not have properties, so the addition of properties would be another obvious extension.

There currently does not exist a widely used implementation of bigraphs so therefore we will have to provide this ourselves.

### 4.5 Data structure comparison

As mentioned above, XML is not really a data structure but it is included here because of its inherent hierarchical structure. XML is an obvious candidate as the data representation of whatever data structure is chosen for the framework. Although XML is not as well established as a data storage technology compared to relational databases, it is widely used as a data transfer technology. Therefore, it would certainly make sense to define an XML format for the framework taxonomies to be used e.g. when exporting or accessing taxonomies over the Internet.

OWL has a very rich and complex type system for defining ontologies, but the application domain is rather narrow. The taxonomy framework is required to handle very generic hierarchical structures and the data structure of OWL is not suited for this.

Bigraphs do not offer as rich a class/property system as OWL, but they do offer composition and reaction rules which can be utilized to manipulate the structure of data. As shown with the use cases this can be employed in many different scenarios.

Bigraphs are very generic in nature and this fits well with the generic nature of the taxonomy framework. It also means that it will be easy to extend the existing bigraph theory and create e.g. a type system that better fits the taxonomy framework.

In conclusion, bigraphs are the most suitable of the data structures surveyed here because of their generic nature, which means they can be easily extended, and because of the functionality provided by composition and reaction rules.

We can, however, look to OWL for inspiration when designing the type system of the taxonomy framework. In OWL properties are typed and thus it is possible to restrict what objects can be related to what other objects. The type system of the taxonomy framework will be designed to provide similar functionality (see next section).

Were we to fully implement the framework we could also provide OWL as an export format, in which case the framework could be used to build an OWL editor.

The framework prototype we intend to implement will be based on bigraphs, and therefore we will have to provide the extensions to the theory discussed above. We will discuss this in more detail in the next section.

## 4.6 Extending bigraphs

Although the Taxonomy Tool and bigraphs are very similar in that they both utilize hierarchical data structures with links between them, there are also some differences. For example, in bigraphs all data is contained within the system, whereas nodes in the Taxonomy Tool often reference data outside of the system. Therefore, it will be necessary to extend the functionality provided by bigraphs to achieve the requirements specified in chapter 2.

In the following sections we will describe the areas where extensions are necessary.

### 4.6.1 Bigraph element names

Many of the bigraph components, such as the bigraph itself, nodes and edges must have names. Names add semantic value to these components, and this improves the usefulness of the structures. If the components do not have names, users will not be able to differentiate between them.

The names of components, however, will have no influence on the structure of the bigraphs the components are part of and they are not synonymous with identifiers in this context.

### 4.6.2 Types

Nodes in the Taxonomy Tool and nodes in bigraphs are typed in slightly different ways. In the Taxonomy Tool, the type of a node decides how that node

references external data and in bigraphs a node is typed by its control, which, through its ports, controls how a node can be linked to other nodes.

When using bigraphs in the taxonomy framework these differences are best handled by moving the type information, including the external reference (see section 4.6.4), to the control. In this way we keep all type information in one place. Therefore, the controls will be extended to provide this extra information.

Furthermore, we would like to expand the notion of types to provide the possibility of building even more advanced rule sets for composing and linking graphs as mentioned in section 2.4. We will let ourselves be inspired by the OWL type system when designing this extension.

The OWL type system mainly affects how objects are horizontally related. It is possible to type relations (or properties) and thus restrict what types of objects can be related.

In the context of bigraphs this could be achieved by typing ports and links and enable the definition of restrictions as to what ports can be connected to what links.

Milner has furthermore suggested defining restrictions on what types of nodes a node can have as children, based on the controls of the nodes.[5]

Implementing this functionality as an extension to the bigraph implementation would equip the Taxonomy Tool with a rich type system, which would satisfy the requirement mentioned above.

For example, we might imagine a form control for nodes representing forms in the form taxonomy and a law control for nodes representing laws (see the use case in section 2.2.1). A formToLaw edge would be defined to represent a relation between a form and a law, showing what law necessitates a form. Such an edge would be restricted to connect to exactly one formToLaw port (on the form node) and one lawToForm port (on the law node).

Furthermore, a control representing government authorities would be restricted to only having nodes with form controls as children. Thus we have defined restrictions on horizontal as well as vertical relationships.

### 4.6.3 Node properties

As specified in section 2.4 it should be possible to define properties on nodes. This is not present in the theory of bigraphs, and will therefore have to be implemented in the extension of the bigraph implementation. In this context we define properties as simple key/value pairs, where the value always is a string.

Properties will enable the user to attach meta data to taxonomies that do not contain external references, or meta data that simply does not belong outside of the taxonomy.

For example, the forms taxonomy mentioned above is used in several projects, and each project might define properties that only relate to that project. This could be a flag indicating whether a form has been processed or a free text comments field.

### 4.6.4 External references

It should be possible to define external references on nodes. An external reference is essentially a reference to some external data source, where meta data about that node is stored.

External references can reference a variety of data sources, e.g. databases and XML documents, so it will be necessary to define different external reference types.

The external references of nodes of the same control should reference the same data source. Therefore, the external reference of a node should be defined by the node's control. This will be done by creating an external reference definition and coupling it to a control. Since different external reference definitions can have the same type, e.g. database, each external reference will reference an external reference type.

As mentioned above, the data source of an external reference should be defined on the control, ensuring that all nodes with the same control reference the same external data source. Each node should then have a unique identifier indicating what data in the data source is related to that node.

For example, the form control would point to the database from which the forms taxonomy was originally generated and each node referencing the form control would carry a unique identifier pointing to exactly one row in that database. Thus data about the form which the node represents can easily be accessed.

To summarize, we will define a number of external reference types, e.g. database and XML based external reference types. Users will then be able to define external reference definitions which each have an external reference type. Controls can be coupled to the external reference definitions. Finally, a node can have an external reference ID which, coupled with the external reference definition of the node's control, gives access to the data of that node.

#### 4.6.5 Persistence and sharing

Bigraph theory is mathematical theory and therefore does not concern itself with persistence of data. In our framework, however, this will be an important consideration.

When working with taxonomies it is often useful to work with several different taxonomies, and create relationships between the nodes in the taxonomies (see the forms use case in section 2.2). Often, one central taxonomy, e.g. the forms taxonomy, will be coupled with many different taxonomies depending on the project at hand.

In bigraph theory, it is possible to create links between nodes in different regions, but the regions have to be in the same bigraph. This implies that if we want to work with two taxonomies at the same time, they should be regions in the same bigraph.

In the case of the forms taxonomy, with all the taxonomies it has been related to, this could grow to become a rather large bigraph, and many of the regions in that bigraph would be completely unrelated. Additionally, if some of those taxonomies had also been related to other central taxonomies, they would also need to be included in the same bigraph along with all the other minor taxonomies they had been related to. Consequently, this could grow to become a massive and rather unwieldy structure.

Another issue relating to persistence is that of sharing of bigraphs. When two bigraphs are composed in bigraph theory they cease to exist as individual bigraphs. This means that sharing a bigraph by composing it with several other bigraphs is impossible, unless the bigraph is first copied and then composed.

But copying means that future changes in the original bigraph will not have effect on the copies, and this is counterintuitive to the concept of sharing.

We have therefore come up with a way to solve both of these problems that we consider acceptable within the domain of bigraph theory. We will define a special type of bigraph, a *composite bigraph*, that which contains references to a number of *component bigraphs*. Thus, the composite bigraph contains no nodes of its own.

Links between component bigraphs will be created as similarly named outer names in each of the participating bigraphs. The composite bigraph is then the result of parallel horizontally composing the component bigraphs. As the component bigraphs are parallel horizontally composed matching outer names will be merged as edges during this operation.

A composite bigraph will define its own set of outer names, but the inner names and the outer width and inner width will be respectively the combined inner names of the component bigraphs and the total of the outer widths and inner widths of the component bigraphs.

Sharing will be made possible by building bigraphs as composite bigraphs. For example, workflows could be built as composite bigraphs, making possible the definition and sharing of single workflow steps as prime bigraphs.

Figure 4.7 illustrates the principles of composite bigraphs. The composite bigraph  $GH$  consists of the individual bigraphs  $G$  and  $H$ .  $GH$  is a bigraph with three regions each containing one hole, i.e. a *merge* building block.  $GH'$  represents the composite bigraph after the individual bigraphs is composed into it.  $G$  and  $H$  each have an outer name  $y$ , which are merged into an edge when the component bigraphs are parallel horizontally composed into  $GH$ . Here we see how the common outer names in the component bigraphs represent the links that exist between the individual bigraphs.

#### 4.6.6 Extension formalization

In this section we will present a formalization of each of the proposed extensions. The reader should refer to the previous sections for the explanations of each extension.

**Definition 4.6.1** (bigraph element names). We define the name map  $name : E \cup V \rightarrow N$  where  $N$  is a set of names. This map will provide a name for the nodes and edges. Furthermore we define a bigraph name  $n \in N$  on the bigraph.

The bigraph will thus be defined as  $G = (V, E, N, ctrl, name, G^P, G^L, n) : I \rightarrow J$

**Definition 4.6.2** (types). In definition 3.2.1 the signature is defined as a set  $\mathcal{K}$  of controls. We will change this so that a signature is no longer a set, but a tuple of sets.

We define a set of *controls*  $\mathcal{K}$ , a set of *port types*  $\mathcal{PT}$ , a set of *link types*  $\mathcal{LT}$  and a set of *external reference definitions*  $\mathcal{ER}$ .

For each control  $K \in \mathcal{K}$  the signature provides a finite ordinal  $ar(K)$ , an *arity*.

The signature furthermore provides the mappings  $allowableChildren : \mathcal{K} \rightarrow \mathcal{P}(\mathcal{K})$ ,  $allowablePortTypes : \mathcal{LT} \rightarrow \mathcal{P}(\mathcal{PT})$  and  $extRefDef : \mathcal{K} \rightarrow \mathcal{ER}$ .

We can then formally define the signature  $S = (\mathcal{K}, \mathcal{PT}, \mathcal{LT}, \mathcal{ER}, ar, allowableChildren, allowablePortTypes, extRefDef)$ .

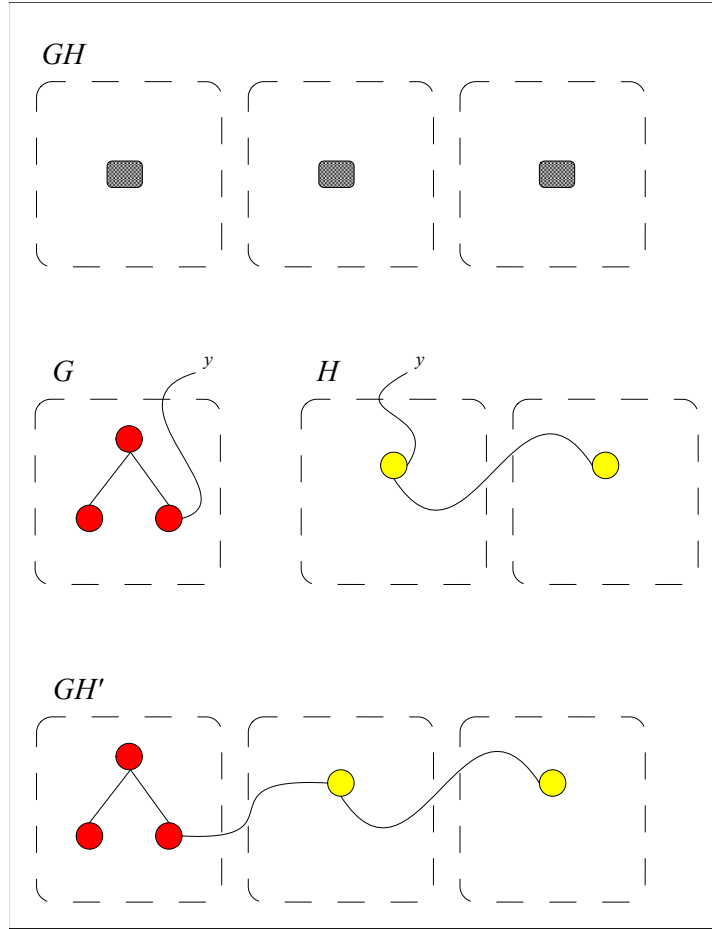


Figure 4.7: A composite bigraph consisting of two individual bigraphs

**Definition 4.6.3** (extended placegraph). The place graph provides the port type mapping  $portType : P \rightarrow \mathcal{PT}$ .

The place graph provides the property mapping  $prop : V \rightarrow \mathcal{P}(P)$  where  $P = (Key \times Value)$  is a set of node properties,  $Key$  is a set of keys and  $Value$  is a set of values.

For each node  $v \in V$  the function  $extID(v)$  provides an *external ID* and the mapping  $extRef(extID(v), extRefDef(ctrl(v)))$  provides an *external reference*.

The place graph is thus defined as  
 $G = (V, P, ctrl, prnt, portType, prop, extRef, extID) : m \rightarrow n$ .

**Definition 4.6.4** (extended linkgraph). The link graph provides a link type map  $linkType : E \uplus Y \rightarrow \mathcal{LT}$ .

The link graph is thus defined as  $G = (V, E, ctrl, link, linkType) : X \rightarrow Y$ .

**Definition 4.6.5** (well typed extended bigraph). We define the restriction that the children nodes of a parent node is restricted to nodes of the controls defined as allowable children on the control of the parent node, i.e. for all  $cv \in prnt^{-1}(v)$

it must hold that  $ctrl(cv) \in allowableChildren(ctrl(v))$ .

We define the restriction that links can only be connected to ports of the port types specified by the link's link type, i.e.  $portType(p) \in allowablePortTypes(linkType(link(p)))$  for all  $p \in P$ .

**Definition 4.6.6** (composite bigraph). We assume an indexed set of extended bigraphs  $\mathcal{A} = \{A_i\}_{i \in \mathcal{I}}$ .

A composite bigraph takes the form  $C = \langle i_0, i_1, \dots, i_n \rangle : I \rightarrow J$  for  $i_j \in \mathcal{I}$  where the interfaces  $I = \langle m_0, m_1 + \dots + m_n, X_0, X_1 + \dots + X_n \rangle$  and  $J = \langle n_0, n_1 + \dots + n_n, Y_0, Y_1 + \dots + Y_n \rangle$  are its inner and outer faces. The composite bigraph  $C$  then denotes the bigraph  $A_{i_0} | A_{i_1} \dots | A_{i_n}$ .

As can be seen from the definition, composite links are created by parallel horizontally composing each of the component bigraphs, so that commonly named outer names in the component bigraphs are merged as edges.

Since composite bigraphs are made up of references to component bigraphs, and thus contain no nodes or links themselves we will not define composition for composite bigraphs.

**Definition 4.6.7** (taxonomy system). This last definition we will provide to clarify the relationship between (extended) bigraphs and composite bigraphs and the notion of sharing of bigraphs.

A taxonomy system  $TT = (\mathcal{A}, \mathcal{C})$  contains a set of extended bigraphs  $\mathcal{A}$  and a set of composite bigraphs  $\mathcal{C}$ .

We see that for each composite bigraph  $C \in \mathcal{C}$  it holds that each component bigraph referenced in  $C$  comes from  $\mathcal{A}$ . Since composite bigraphs from the same system consist of references to bigraphs in that system, we see that each bigraph  $A \in \mathcal{A}$  can take part in several contexts, i.e. they can be shared.

## 4.7 Summary

In this chapter we have analyzed existing data structures on which the framework could be based, and discussed advantages and disadvantages of each.

Each of the data structures analyzed has their own advantages and disadvantages. Specifically, OWL offers an advanced type system and bigraphs, which will be employed in the context of this thesis, offer unique features such as composition and reaction rules.

We have formally defined a number of extensions to the existing bigraph theory, that must be included in order for the data structure to satisfy the defined requirements.

In the next chapter we will analyze the architecture of the taxonomy framework, which will be based on bigraphs and the extensions to the bigraph theory defined in this chapter.

## Chapter 5

# Framework Architecture Analysis

In this chapter we will provide an analysis of the architecture of the taxonomy framework. We will discuss how we can implement the framework based on the definitions presented in chapters 3 and 4. In particular, we will analyze the problems in modeling bigraphs in an object-oriented programming environment.

Since we are building a prototype we will not consider all the requirements mentioned in section 2.4. Rather we will concentrate on those requirements which affect the underlying data structure of the framework. This means that such requirements as security and accessibility will not be considered here, although they certainly will be important requirements in a production system.

### 5.1 Overall framework structure

We will now begin analyzing how to structure the bigraph-based taxonomy framework in order to achieve the requirements specified in chapter 2.

The requirements can be achieved with several different architectures, but we will attempt to follow established practices in order to achieve a functional design.

#### 5.1.1 Framework structure

It is a stated requirement that the framework should have an API, so that it can be easily utilized by developers. It is also a stated requirement that it should be possible to persist taxonomies in different data representation formats such as relational databases and XML.

When working with different data representation formats it is desirable to achieve a high level of transparency, because then users will not have to worry about customizing applications for specific formats. For example, a person could write a program that handles taxonomies stored in relational databases as well as XML.

To ensure that the underlying data format is transparent for the user, we will base the design on a three-tiered design commonly seen in so-called enterprise architectures. The framework will then consist of three layers: the framework

core (also called the business layer), the data access layer (DAL) and the data layer (also called the persistency layer).

Figure 5.1 illustrates this design. The uppermost tier represents the framework core which accesses the middle tier, the DAL. The DAL defines the methods with which the framework core accesses data from each of the data sources in the bottom tier. In this diagram there are two data sources: a relational database and an XML document.

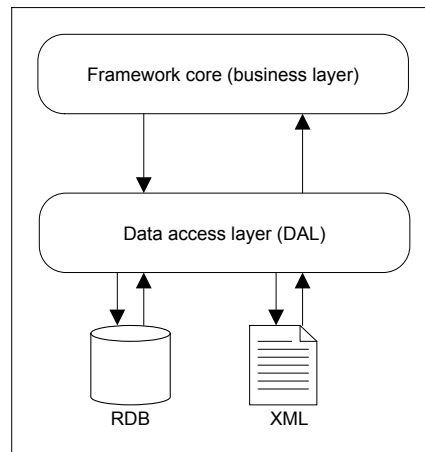


Figure 5.1: The three layers of the framework: the framework core, the data access layer and the data layer

The framework core derives its name from the fact that it is at the center of the framework. On top of the framework core there might be an extra tier for availability considerations, e.g. a web service that publishes an instance of the framework to the Internet, or some user application which consumes the framework. Beneath the framework core there will typically be a data access layer and one or more data layer instances.

The framework core will contain the logic that models bigraphs, and it can be used with or without the data access and data layers. If used without, the structures created will only exist in memory and will not be persisted.

The framework core will define how composition and reaction rules are handled, and it will contain logic for communicating with the data access layer, so that consuming applications will only have to access the framework core.

Since we want to be able to store bigraphs in different formats, eg. relational databases and XML, the data access layer will define an interface for accessing and persisting bigraphs that are stored. We will then provide implementations of this interface for each storage format along with the data representation format definitions. For example, if we want to be able to store bigraphs in relational databases as well as in XML documents we will have to define a relational database schema and an XML schema, and provide a data layer access instance for each of these.

This is a flexible design that allows for extensions of the framework core, if a user requires some functionality that is not already provided by the framework (or if a user wants to restrict the functionality of the framework). A user can simply define a new layer on top of the framework core that adds the desired

functionality. It also allows for data storage in any format required by a user. The user will just need to implement a DAL and a data layer representation format which satisfies his requirements.

## 5.2 Framework core

We have already established that we will employ the bigraph model presented in chapter 3 as the underlying data format in the framework, and that we will have to extend this model to meet all the defined requirements.

Since we are making an implementation based on current bigraph theory, we choose to also make this available on its own without extensions so that users can use this if they want to use a data structure that is closer to bigraph theory. We will then define the extensions in a separate library on top of the base bigraph implementation. This way we will have two data structures that can be employed for different purposes.

Because we are basing the data model on a well founded mathematical model, we want the implementation to remain as faithful to the mathematical definitions as possible. In some cases, however, major performance gains can be achieved or usability can be improved by implementing details in an interpretive manner.

For example, as we shall discuss later, implementing reaction rules according to the definitions in chapter 3 is very complex, but if we provide the same functionality through different means that are simpler to implement, e.g. instance methods, we do not violate the theory. We simply interpret the instance methods to implement the reaction rule definitions. Of course, we will not add functionality not defined by the theory in this way.

In our implementation we will only provide DAL and data layer implementations for the bigraph extension implementation. This means that it will not be possible to persist bigraphs created with the base bigraph implementation.

Figure 5.2 illustrates the design that is described here. The base bigraph implementation implements the theory described in chapter 3, and the bigraph extension implementation implements the extensions described in section 4.6.

### Identifiers

Because we will also be implementing a data layer it will also be necessary to add machine understandable unique identifiers to all components.

This is not necessary when reading the data from a data source, but when writing the data back it is critical that we know exactly what piece of data corresponds to each component in the bigraph. Otherwise, updates will not be possible as we won't know what to update.

Therefore all components will be equipped with a unique identifier which will be a 16 byte randomly generated string. This is a small extension to the theory that is made necessary by the requirement of persistence, and it will apply to the base bigraphs as well as the extended bigraphs.

Although this is in fact a theory extension we will not formalize it as it represents meta data that is made necessary by the persistence requirement which is not covered by the theory either.

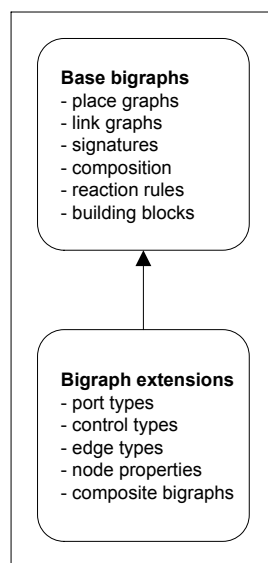


Figure 5.2: The bigraph implementation structure

### Compile time type definitions vs. runtime type definitions

In the following sections we will discuss two different kinds of type definitions.

*Compile time type definitions* are what is normally understood by the word “type” in an object oriented programming language context, i.e. class definitions.

*Runtime type definitions* is the term we use for types that users of the framework should be able to define at runtime. These include controls, port types and link types. These components are represented by class definitions, and each user defined type is represented by a class instance.

For example, in the case of link types we will define a LinkType class, and each link type definition will be an instance of that class. Therefore the LinkType class will be a compile time type and each LinkType instance will be a runtime type.

### Collection classes

In the following sections we will often use the terms “collection” or “collection class”. These refer to a class which represents a strongly typed collection of instances of some other class.

Collection classes are often used as an alternative to storing objects in arrays, and generally result in less error prone code than when using arrays because of their type-safeness.

Some collection classes are ordered, meaning that objects contained in the classes are sorted according to some sorting scheme. The sorting scheme depends on the implementation of the contained class. If a specific collection class is ordered we will write this, otherwise the reader should assume that the collection is not ordered.

Collection classes provide iterator classes. These are based on the Iterator

design pattern from [1], and enable customized iteration through a collection of objects.

During the analysis we will refrain from going into detail with all collection class implementations, and instead assume that the reader understands the basic functions of these classes.

### 5.2.1 Base bigraphs

We will now begin analyzing the design of the base bigraph implementation.

Bigraphs consist of many different components, and we will discuss how each of them is best handled in the framework. We will take a top-down approach by first discussing the overall bigraph structure and then moving on to individual components.

First we will discuss an existing model for working with hierarchical structures, namely the Document Object Model, on which we have chosen to base the bigraph implementation.

#### The Document Object Model

At its most basic a bigraph is a hierarchical structure. The links in bigraphs are not hierarchical in nature, but they have little significance if they have no nodes to connect. When designing the base bigraph implementation, we therefore think of the data structure as first and foremost a hierarchical structure.

To gain inspiration for designing such a structure we have looked at a different implementation of tree structures, namely that of the Document Object Model (DOM) [12].

The DOM is a platform- and language-neutral specification of an API for accessing and manipulating hierarchically structured documents programmatically. Specifications exist for handling XML, HTML and CSS documents.

The DOM models documents by storing references between nodes in a document. In order to do this the entire document must be parsed once, and afterwards kept in memory. This makes the DOM unsuitable for very large structures, but “for applications requiring random access to different portions of a document at different times or applications that need to modify the structure of an XML document on the fly, DOM is one of the most mature and best-supported technologies available” [2].

The use cases in section 2.2 include cases where users access taxonomies through a graphical user interface. In such cases it is impossible to predict what node a user will want to access next. He might want to find a descendant of some node or he might want to change focus to a different hierarchy. We therefore see that the taxonomy framework will have to accommodate random access to nodes.

The framework will also define ways of modifying structures by composition and reaction rules. These are complex operations and may incur heavy processing costs if the underlying data structure does not support it properly.

We can therefore conclude that the base bigraph implementation, in particular when considered in the context of the taxonomy framework, has both the requirements mentioned above, and this indicates that modeling the hierarchical structures in the base bigraph implementation after the DOM model would be a reasonable choice.

A consequence of basing the implementation on this model is that when a bigraph is loaded from its data source the whole structure, including both the place graph and link graph, will be loaded and kept in memory as long as it is needed. This may incur some performance overhead in case of large structures, but we believe that it will improve performance in case of composition and reaction rules and it will improve performance of applications requiring random access, such as graphical user interfaces.

When also considering the defined bigraph extensions, we should remember that the data content of nodes will be stored externally and kept as external references in the nodes. This will help improve performance in case of large structures, because the amount of data it will be necessary to keep in memory will be smaller. This data will only be loaded when requested by the user, and will therefore not have influence on the performance of actions regarding the structure of the data.

We will not provide a full implementation of the DOM interface as this is rather complex. Rather we will let our design be loosely inspired by the DOM, meaning that we will implement a subset of the interface and general design principles will be guided by the design of the DOM. For example, we will not treat all entities (e.g. attributes, comments, elements etc.) as nodes, as the DOM does. Rather we will treat only what we call “nodes” as DOM nodes. How each component of the framework is affected by this will be described in subsequent sections.

## Names

The element name extension (see definition 4.6.1), which is the only extension that is included in the base bigraph implementation, will be applied to some of the elements discussed here. This is because this extension does not interfere with the mechanisms defined by bigraph theory. All it does is add a name to some of the components, e.g. the bigraph component and the node component.

In all cases the extension will be implemented as a simple text property.

## Bigraph

Although this is arguably the most important component in the structure, it does not contain a lot of information. It acts as a container that keeps all the other components together, and therefore it will contain references to these.

However, keeping references to all components in the bigraph component is unwieldy. Bigraphs contain a place graph and a link graph, and by defining similar components in our structure we can separate tree information from link information.

Thus the bigraph component will only contain references to a place graph and a link graph and not to their individual components. Each of these components will then keep references to the individual nodes and edges of the bigraph. Additionally, the bigraph component will keep a reference to a signature component containing control definitions.

This will also be the case for bigraph interfaces, which are tuples consisting of a width and a name set (see definition 3.2.2). Since the information contained in an interface originates from the place graph and link graph, we will only

make the information available there, and thus we will not make individual components representing interfaces.

It will not make sense for a place graph or even a node to exist on its own, outside of the context of a bigraph. If this happens the outcome is unpredictable because the dependencies of that place graph or node are not satisfied. For example, what should happen if a user tries to instantiate a node which doesn't belong to any bigraph? This is not permitted in bigraph theory and clearly such a situation should not be allowed to happen.

To avoid such situations individual components will not have external constructors. They will all be constructed through factory methods on the bigraph, place graph and link graph components, which will ensure that they are associated with the right bigraph.

### Place graph

A place graph contains information about the trees that bigraphs are made up of. As the place graph is the only component of the bigraph to contain hierarchical information, it is also the design of the place graph that will be most visibly inspired by the DOM.

The DOM represents a hierarchical structure by equipping each node with a parent property pointing to another node. This ensures that all nodes have only one parent. The DOM furthermore specifies that it is illegal to create circular references.

The place graphs in our implementation will be built on the same principles, meaning that nodes will be represented by node objects and each node object will point to its parent which will also be a node object. Since place graphs are made up of regions, nodes and holes we will have three different kinds of node objects.

We want to ensure that the classes we define resemble their theoretical counterparts. The place graph structure defined in chapter 3 looks like this:  $G = (V, ctrl, prnt) : m \rightarrow n$ . It consists of a set of nodes  $V$ , a set of controls  $ctrl$ , a parent mapping  $prnt$  and an inner width  $m$  and an outer width  $n$ .

The parent mapping is defined as  $prnt : m \uplus V \rightarrow V \uplus n$  and  $prnt^k(v) \neq v$  for all  $k > 0$  and  $v \in V$  (see definition 3.2.4). This means that only regions and nodes can be parents and only nodes and holes can be children and that the place graph must be acyclic.

The set of nodes  $V$  and the parent mapping  $prnt$  will implicitly be present, because these represent the nodes and the tree structure which is defined by the node objects. The control set  $ctrl$  will also be made available implicitly, because each node will reference a control from the control set. The inner width and outer width can be calculated by counting the number of holes and regions in the place graph, but this may become a costly operation so they will also be kept as properties on the place graph to improve performance.

Furthermore, regions and holes should be ordered. In bigraph theory regions as well as holes have an implicit ordering, which is important when vertically composing bigraphs. Therefore each place graph will have an ordered collection of regions and an ordered collection of holes referencing respectively the regions and the holes in the place graph tree.

The final design should reflect the mathematical definitions while also conforming to the DOM. We will now analyze how each of the place graph compo-

nents defined by the theory will be handled in the implementation so that this requirement is fulfilled.

A place graph consists of zero or more regions. A region may contain several trees. Therefore a region can be thought of as a tree in itself with the region as the root node, and each contained tree as a subtree of the region. The region node will simply be a special kind of node that has no parent and no control. Treating the whole region as a tree, will make it easier for us to base it on the DOM, because the DOM treats all entities as nodes. In some bigraph definitions regions are actually treated as root nodes (e.g. in [4]), so this is in line with the established theory. A region cannot be attached to an edge like a node, because it has no control and therefore no ports.

We could also consider treating the whole place graph as a tree, with a place graph node as the root node and a number of region nodes as its children nodes. However, it is more consistent not to do this and simply have a place graph class that references the individual regions. This is because we want the place graph to structurally resemble the link graph, in the same way that the two graphs are rather similar in bigraph theory, and then it makes less sense to design the place graph as being part of some graph structure.

This design will have an influence on the ordering of the regions and holes mentioned above. Since each region is the root node of a tree the region collection will actually be a collection of trees, and since each hole will be part of a tree all holes will be accessible through the region collection as well as the holes collection. However, the order of the holes will only be defined by the holes collection, meaning that the tree structures has no impact on this.

Regions can contain both nodes and holes. Therefore we will have three different types of tree entities: nodes, holes and regions. We have already discussed the region entity, so we will now proceed to discuss the two others.

Holes are similar to regions in that they have no controls, and therefore cannot be attached to edges. Holes cannot have children and they must have exactly one parent which is either a region or a node.

A node will contain a reference to its control (see section on signatures on page 55). It will also contain a collection of ports. Each port may be connected to a link (an edge or an outer name) indicating that the node is connected to that link through that port.

In the DOM all DOM nodes implement the same interface `Node`. This interface provides methods for iterating through and manipulating the hierarchical structure. In the same way region, node and hole implementations will implement the same DOM-inspired interface in the framework.

Iteration is achieved by providing access to the children of a node. In this way it is possible to move through the tree until no nodes with children are left, in which case the bottom of the tree has been reached.

Manipulation of the tree structure is achieved with the implementation of a subset of the properties and methods defined in the DOM interface. The methods we have chosen to implement are listed in figure 5.3.

We have chosen a minimal implementation in the sense that we have only included the properties and methods that are absolutely necessary to provide a functional API.

When appending a node to another node it is important to maintain the restraint that parent mappings be acyclic. Practically this means that a node cannot be its own ancestor.

Name	Description
parentNode	Returns the parent of a node
childNodes	Returns the child node collection of a node
replaceChild	Replaces a child node with a different node
removeChild	Removes a child node from a node's child node collection
appendChild	Adds a child to a node's child node collection
hasChildNodes	Returns true if a node has child nodes

Figure 5.3: Dom interface properties and methods included in the Bigraph implementation.

We have left out sibling references, because they tend to take up a lot of memory and we consider them convenience methods that are less useful in this context than when manipulating XML or HTML documents, which is what the DOM was originally designed for.

We have left out all the properties and methods that has to do with node values and attributes, because bigraphs have no notion of these.

We have left out properties and methods that has to do with the ordering of nodes (such as `firstChild`). This information will instead be included in a node's child node collection which will be an ordered collection.

The methods defined by the DOM interface are clearly not covered by the definitions in chapter 3, yet the functionality the methods provide can also be provided by the theory in the form of reaction rules. We will discuss this in more detail in the section on reaction rules on page 59.

### Link graph

The mathematical definition of a link graph is:  $G = (V, E, ctrl, link) : X \rightarrow Y$ . Here  $V$  is a set of nodes,  $E$  is a set of edges,  $ctrl$  is a mapping from nodes to controls,  $link$  is a mapping from points (ports and inner names) to links (edges and outer names).  $X$  and  $Y$  are the inner face and outer face.

$V$  and  $ctrl$  also participate in the definition of place graphs, and logically belong in the place graph. They are included in the definition because the  $link$  mapping use information from them. However, we will only make these available through the place graph, although the link graph will contain internal references to these.

The theory defines the link mapping as  $link : X \uplus P \rightarrow E \uplus Y$  where  $P \stackrel{def}{=} \sum_{v \in V} ar(ctrl(v))$  is the set of *ports* of  $G$ . This means that connections go from inner names and ports (i.e. points) to edges and outer names (i.e. links) and each port and inner name can be connected to exactly one edge or outer name.

The  $link$  mapping will be implemented by collections on the objects that represent the links, i.e. the edges and outer names. These will be instances of classes that implement a link interface. The collections will contain references to objects representing points (and implementing a point interface), and each reference will represent a connection. It will be ensured that all such collections are disjoint, i.e. that no two collections contain a reference to the same port.

This means that the implementation will work a little differently from the mathematical definition, in that the mathematical definition is a point-to-link

function whereas the implementation is a link-to-point function (see definition 3.2.6). However, because we ensure that the collections are disjoint the end result is the same and it is therefore an acceptable change.

Edge and outer name classes will define functionality for creating and deleting connections (by respectively adding a point object to or removing a point object from the point collection of the edge or outer name).

Furthermore, the edge and outer name classes will define functionality for merging links. This is necessary when performing vertical and parallel horizontal composition. For example, during vertical composition it might be necessary to merge an outer name with an edge connected to an inner name. In this case all the connections belonging to the outer name is moved to the edge and the outer name is deleted. During parallel horizontal composition it might be necessary to merge two outer names. In this case all the connections on one of the outer names is moved to the other outer name and the outer name with no connections left is deleted.

Both the functionality of creating and deleting connections and merging links is covered by the theory so we do not need to define it further. The methods are listed in figure 5.4.

Name	Description
AddPort	Creates a connection to a port
RemovePort	Deletes a connection to a port
Merge	Merges the link with another link

Figure 5.4: The link graph API.

### Signature

The theory defines a signature as a set of controls. For each control  $K$  the signature provides a finite ordinal  $ar(K)$ , an *arity* (see definition 3.2.2). Furthermore, the control map is defined as  $ctrl : V \rightarrow K$ , meaning that each node references one control (see definition 3.2.3).

In our implementation we will provide a control class and each instance of this class will represent a control. Such a control instance will have an arity, which is a number, and each node instance will contain a reference to exactly one control instance. The sum of all such references will represent the *ctrl* mapping mentioned in the bigraph definition in chapter 3. We will not provide access to this mapping as a collection, however, since each mapping is only useful in the context of the node involved.

Since controls contain type information for nodes it can be said that control instances are runtime type definitions (see section 5.2).

### Composition

We recall from chapter 3 that there are two basic categories of composition: vertical composition and parallel composition.

Vertical composition is the operation of inserting or 'plugging' one bigraph into another bigraph. This is the most complex of the two categories of composition, because the inner and outer faces of the two bigraphs must match. Additionally, it is required that the node sets and edge sets of the two bigraphs

are disjoint. In the literature vertical composition is denoted by the binary  $\circ$ -operator.

Horizontal composition is basically the operation of merging two bigraphs by placing their individual components next to each other, but several variations of this are defined.

Tensor horizontal composition is the operation that is included in the definitions in chapter 3. This is the most commonly referred to variation of horizontal composition and in the literature it is denoted by the binary  $\otimes$ -operator. Tensor horizontal composition requires the two bigraphs to have disjoint node sets and disjoint inner and outer name sets.

Parallel horizontal composition has not been covered by the theory presented in chapter 3, but we include it here because we use it for composite bigraphs. Parallel horizontal composition is similar to tensor parallel composition, except that if the two bigraphs have the same name in the outer face, the names are merged into one name. Thus it is not necessary for the two bigraphs to have disjoint outer name sets. Parallel horizontal composition is denoted by the  $|$ -operator.

As we can see from above review of bigraph composition variations, disjoint node sets and disjoint name sets is a common requirement. This is due to the way identity of components is defined in the theory. If two nodes have the same identifier, they represent the same node, even though they may exist in different node sets. Similarly, if two names are the same, they represent the same component. This is because after composition it will not be possible to differentiate between the two nodes or names.

In bigraph theory, this issue is solved with “abstract bigraphs” (see definition 3.2.9). Abstract bigraphs are concrete bigraphs where the identities of nodes and links are abstracted away. This makes composition of bigraphs with non-disjoint node sets and/or name sets possible. Concretely, this is usually implemented by renaming conflicting names during composition.

So these requirements exist because we cannot allow identity conflicts in the bigraphs. When modeling bigraphs in an object oriented programming language, however, we have to consider that the concept of identity works differently than in mathematical theory.

In a runtime environment the identity of an object, i.e. an instance of a class, is the address in memory at which that object is stored. In our bigraph implementation all references will be object references and thus the runtime environment will ensure that identity conflicts do not occur. Therefore node sets will always be implicitly disjoint.

The same principles apply to edges as do to nodes. In our bigraph implementation the identity of an edge is defined by its instantiated object, and an edge can only exist within the context of one bigraph at a time. Therefore there is no need to worry about non-disjoint edge sets either.

Outer and inner names work differently from nodes and edges in that they are identified by their names as they are in bigraph theory. This is because, unlike holes, outer and inner names are not ordered, and therefore, the only way to connect an outer and an inner name is through a common identifier, i.e. the name. Thus, during tensor horizontal composition it will still be necessary to ensure that the outer name sets and inner name sets of the two bigraphs are disjoint.

For our implementation this is a better solution than using abstract bigraphs,

because we avoid renaming components during composition. If a node or an edge has been named manually by a user, it is likely that the user will not want that name to change by itself.

During composition the elements of one bigraph is moved to another bigraph, and the bigraph whose elements have been moved subsequently ceases to exist. Therefore we will implement composition functionality as methods on the bigraph instances. The bigraph instance on which a composition method is called will then be the bigraph into which the elements of the other bigraph instance is moved. The other bigraph instance will be provided as an argument to the method call, and this instance will be deleted after the composition is completed.

For the composition of two bigraphs  $A$  and  $B$  to be possible they must have matching signatures. Although reasonable in theory, this restriction seems to be unnecessary in practice and will only add manual work. Therefore we have decided to add whatever controls are in  $B$  to the signature of  $A$ . This has the same effect as manually adding the controls and then composing, which would have been necessary to do if we had followed the theory strictly on this point. This applies to both vertical and horizontal composition.

### Vertical composition

Vertical composition of two bigraphs is a complex procedure. To compose a bigraph  $G_0$  into a bigraph  $G_1$ , the outer face of  $G_0$  must match the inner face of  $G_1$ . For the inner face of  $G_0$  to match the outer face of  $G_1$ , it is necessary that:

- the number of regions in  $G_0$  equal the number of holes in  $G_1$ .
- the outer names in  $G_0$  equal the inner names in  $G_1$ .

Before vertical composition can take place, it should therefore be checked if it is at all possible to complete that composition. If it is not possible, the operation must be aborted.

Methods for checking if vertical composition is possible will be made available on the bigraph component. As we have discussed above, it will not be necessary to ensure that node sets and edge sets are disjoint, because this requirement is implicitly fulfilled.

Vertical composition of two bigraphs can be partitioned into two operations: vertical composition of the place graphs and vertical composition of the link graphs. The method call structure will reflect this, meaning that the method which the user calls will be defined on the bigraph component, but this method will call a method on each of the place graph and link graph components, which will handle the actual logic of composition.

Vertical composition of two place graphs is defined as  $G_1 \circ G_0 \stackrel{def}{=} (V, ctrl, prnt)$  where  $V = V_0 \uplus V_1$ ,  $ctrl = ctrl_0 \uplus ctrl_1$ , and  $prnt = (\text{id}_{V_0} \uplus prnt_1) \circ (prnt_0 \uplus \text{id}_{V_1})$  (see definition 3.2.5).

This means that the node sets, the control mappings and the parent mappings of the two place graphs are merged. When merging the parent mappings the holes of  $G_0$  are replaced with the children of the regions of  $G_1$ .

Since the parent mappings are stored in the individual nodes in our implementation, vertically composing two parent mappings will simply be a matter of

setting the parent of the nodes to be inserted (the children nodes of the region nodes in  $G_0$ ) to the parent of the matching hole.

Vertical composition of two link graphs is defined as  $G_1 \circ G_0 \stackrel{def}{=} (V, E, ctrl, link)$  where  $V = V_0 \uplus V_1$ ,  $ctrl = ctrl_0 \uplus ctrl_1$ ,  $E = E_0 \uplus E_1$  and  $link = (ld_{E_0} \uplus link_1) \circ (link_0 \uplus ld_{P_1})$  (see definition 3.2.7).

We have already covered composition of the node sets and parent maps and this will be carried out in the context of the place graph. The edge sets and the link mappings should be merged in the context of the link graph. Composition of the link mappings is defined as:

$$link(p) = \begin{cases} link_0(p) & \text{if } p \in X_0 \uplus P_0 \text{ and } link_0(p) \in E_0 \\ link_1(x) & \text{if } p \in X_0 \uplus P_0 \text{ and } link_0(p) = x \in X_1 \\ link_1(p) & \text{if } p \in P_1. \end{cases}$$

This should be interpreted as:

- all edges in  $G_0$  that are connected to an inner name or port in  $G_0$  are left unchanged
- all outer names in  $G_0$  should be connected to the corresponding inner name in  $G_1$
- all links in  $G_1$  that are connected to a point in  $G_1$  are left unchanged

When an inner name and an outer name are connected one of two scenarios can occur. If the inner name is connected to an edge, that edge is merged with the outer name. If the inner name is not connected to an edge a new edge is created which is given the same name as the inner and outer names. If any points were connected to the outer name prior to composition they will be connected to the edge instead.

### Tensor horizontal composition

Tensor horizontal composition of two bigraphs  $G$  and  $H$  requires that:

- $G$  and  $H$  has disjoint node sets.
- $G$  and  $H$  has disjoint edge sets.
- $G$  and  $H$  has disjoint outer and inner name sets.

As explained above, the first two requirements will always be met implicitly. It will, however, be necessary to check that the third requirement is satisfied.

Given the bigraphs  $G : \langle k, W \rangle \rightarrow \langle \ell, X \rangle$  and  $H : \langle m, Y \rangle \rightarrow \langle n, Z \rangle$  tensor horizontal composition of two place graphs is defined as  $G \otimes H : k + m \rightarrow \ell + n$  and tensor horizontal composition of two link graphs is defined as  $G \otimes H : W \uplus Y \rightarrow X \uplus Z$  provided that the previous conditions are met (see definitions 3.2.5 and 3.2.7).

Compared to vertical composition, tensor horizontal composition is a rather simple operation. The regions of each place graph are simply put next to each other, and the holes collections are combined into one. Also, the two link graphs are combined into one without merging any links.

### Parallel horizontal composition

Parallel horizontal composition will function just like tensor horizontal composition, except that names in the outer faces of the two bigraphs that are the same, will be merged into one name.

### Reaction rules

There are basically two ways that reaction rules can be implemented: either as an implementation that follows the principles laid out by the theory so that bigraphs are matched with a redex and transformed according to a reactum or as an interpretive implementation which provides the same functionality as the definitions but without actually doing any matching. These two ways of implementing reaction rules are not mutually exclusive.

In our implementation we have decided not to do a full reaction rule implementation, but instead provide the functionality through different means. We will, however, provide interfaces for a future implementation of reaction rules.

The most important reason for this is that fully implementing reaction rules is very complex. Other theses have been written on this (eg. [13]) although we know of no full implementations.

The main issue when considering a full implementation of reaction rules is the matching of the bigraph structures. Writing an algorithm that finds matches of a place graph within another place graph is in itself not a big issue, but if either place graph contains holes it can become very complex. In reaction rules, holes act as wildcards, i.e. they will match any structure. This complicates the matching algorithm considerably. Above mentioned thesis solved this problem by not allowing sibling holes, which somewhat reduces the complexity.

The interpretive implementation we will provide will be based on the methods defined in the DOM interface. Since the methods will be available on all node instances, each invocation of a method that belongs to a node instance can be said to reflect a reaction rule.

The theory defines the relation between a bigraph  $A$  and a matching redex as  $A = D \circ r$ ,  $D$  being the evaluation context and  $r$  being the redex, and the result of applying the reaction rule as  $A' = D \circ r'$  (see definition 3.2.10).

These relations also apply in the case of our interpretative implementation. For example, the `appendChild` method on an instance of a node reflects a reaction rule which appends a child node to that node. The redex ( $r$ ) of the reaction rule is defined by the node instance and the reactum ( $r'$ ) is defined by a combination of the logic provided by the method (i.e. appending a child node) and the method argument (i.e. the child node to be appended).  $A$  is the bigraph to which the node instance belongs and  $D$  is the part of the bigraph that is not part of the redex.

Interpreting methods that only transform structures as reaction rules is straightforward, however, interpreting methods with return values as reaction rules is rather less intuitive. Reaction rules do not provide facilities for returning values, other than the result of the transformation defined by the reaction rule.

In order to provide reaction rules with return values, we will allow ourselves to expand a little on the theory within the context of the framework.

If a reaction rule needs a return value we will add a region to the redex containing a node with a hole as its only child. We will define a return value

control, which will be the control of the node. In the reactum of the reaction rule the return value, e.g. a node representing one of the boolean values true or false, will then be appended as the child of the return value node.

For example, in the case of the `hasChildNodes` reaction rule the return value region of the reactum will contain a node representing the value true if a node has any children and a node representing the value false if it does not have any children.

This solution, although functional, does indicate that reaction rules are not the perfect theoretical foundation upon which to provide such functionality. An expansion of the reaction rule definitions would be in order for reaction rules to provide a proper foundation for defining generic functionality. It is, however, outside of the scope of this thesis to provide such definitions, and we will instead provide suggestions for future work.

The reaction rule definition for each of the methods provided by the DOM interface (see figure 5.3) is defined in figure 5.5. The bigraphs to the left of the arrows are the redexes and the bigraphs to the right of the arrows are the reactums. The definitions are all provided in the context of the same example node (the leftmost yellow node). The return value node has been given the color blue. The true and false value nodes are marked respectively with a T and an F.

### Building blocks

Building blocks are predefined bigraphs, and as such do not add any functionality to the theory. In the framework we will implement a building block provider, which will have a provider method for each of the six building blocks.

Some of the provider methods will accept arguments specifying the exact structure of the building block to be instantiated. For example, the discrete ion provider method will need to know what control the discrete ion must reference and the merge provider method will need to know how many holes to create. The instantiated building blocks can be used for composition like any other bigraph.

### 5.2.2 Extended bigraphs

In section 4.6 we defined a number of extensions to the bigraph theory from chapter 3 that are necessary for the framework to meet the requirements established in chapter 2. We will now analyze how each of these extensions are best implemented in the context of the previous analysis of the bigraph implementation (section 5.2.1).

The extended bigraph implementation will be a separate class library which extends the classes that are defined in the base bigraph library. Therefore users can decide if they want to use the base bigraph classes or the extended bigraph classes.

### Extended type system

The extended signature is defined as

$S = (\mathcal{K}, \mathcal{PT}, \mathcal{LT}, \mathcal{ER}, ar, allowableChildren, allowablePortTypes, extRefDef)$  (see definition 4.6.2). We will expand the signature to contain a controls set

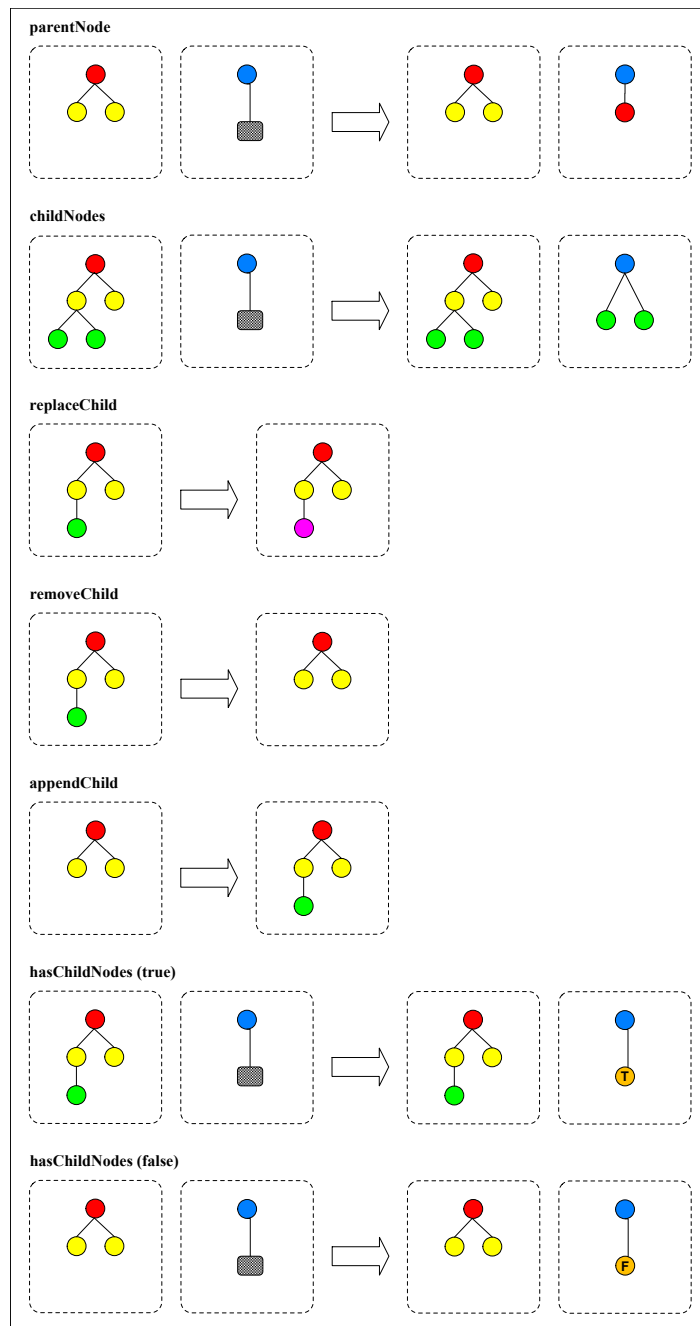


Figure 5.5: The DOM methods defined as reaction rules

( $\mathcal{K}$ ), a port types set ( $\mathcal{PT}$ ), a link types set ( $\mathcal{LT}$ ) and an external reference definitions set ( $\mathcal{ER}$ ).

Link types will have a port type collection indicating what ports links of that type can be connected to. This list will be exclusive, i.e. a link can only be

connected to links of the link types in the list. This collection will implement the *allowablePorttypes* mapping in the extended signature definition.

External reference definitions will contain enough information to access data from a specified data source given a unique identifier. Each external reference definition will also be given an external reference type indicating what kind of data source it references, e.g. a database or an XML document. This, however will not be formally defined as it does not impose any restrictions on the theory. External reference types will be discussed below.

Besides defining how many ports a node has, node controls will also define of what types these ports are. This combined with the port restrictions defined on the link types mentioned above will make it possible to finely control how bigraphs can be structured.

Controls will define what types of nodes a node can have as children. This will be implemented as a collection of controls, which should be interpreted exclusively like the port type collection on the link types mentioned above. This collection will implement the *allowableChildren* mapping.

Additionally, controls will contain a reference to an external reference definition indicating how external data referenced by nodes of that control can be accessed. These references will implement the *extRefDef* mapping.

As is the case with controls, link types and port types are considered runtime type definitions (see page 49).

The *portType* mapping from definition 4.6.3 and the *linkType* mapping from definition 4.6.4 will be implemented by references on the ports and links respectively.

Finally, the methods defined on the points and links and the composition methods (see section 5.2.1) will be implemented so that the restrictions from definition 4.6.5 are not violated.

### Node properties

Node properties are defined as a mapping  $prop : V \rightarrow \mathcal{P}(P)$  where  $P = (Key \times Value)$ .

Node properties will be implemented as simple text based key/value-pairs. Each node will have an unordered collection of properties and the sum of the collections will represent the *prop*-mapping.

### External reference types

We want to be able to reference external data from a variety of data source types, e.g. databases, XML etc. Therefore we will provide an interface for defining external reference types.

It is, however, impossible to predict what information an instance of an external reference type will need to provide access to its data source. For example, in the case of databases a database connection string including logon credentials will be necessary and in the case of XML a URI will be necessary. Other data sources might need more information.

Therefore the interface will only define a name property identifying the external reference type, and the interface implementations will have to define the necessary properties. These implementations will also define the logic for accessing the data sources. Such logic should return the data of one data entity

as string based key/value pairs. This is a very generic format for representing data, and it will be possible to output data from most data sources in this way.

It will be possible to relate a control to an external reference, meaning that all nodes with that control will reference an entity in the data source defined by the external reference. Each node will then have to provide a property that uniquely defines the corresponding data entity in the data source. Since a key in a data source can consist of multiple values, the key/value node properties discussed in the previous section will be used for this. The external reference design is illustrated in figure 5.6.

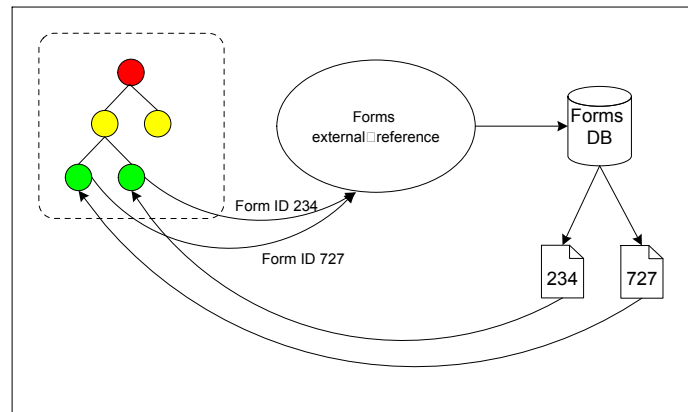


Figure 5.6: An external reference ID combined with an external reference definition points to the external data of nodes.

The *extID* (see definition 4.6.3) function will be implemented as node properties so that each external ID of a node is one or more properties. The name of the properties will be defined by the external reference definition of the node's control. Since the implementation of the *extRef* function (see definition 4.6.3) will vary for each type of data source, this function will be implemented by each of the external reference type implementations.

We will provide one implementation, namely that of database based external resources. As mentioned above, all such an external reference will need to provide is a database connection string and the logic that is used for accessing the database. Given a column key value, this logic will return the data of a single row as key/value pairs, so that each key is a column name and the corresponding value is the value of that column formatted as a string.

It will then be possible to define external references to databases by providing an instance of the database external reference implementation with a database connection string and relating it to one or more controls. Nodes related to the external reference through their controls can then be related to a row of data in the database by providing them with a key defined as one or more node properties.

The functionality described in this section will implement the *extRef* mapping from definition 4.6.3.

### Composite bigraphs

A composite bigraph is defined as  $C = \langle i_0, i_1 \dots, i_n \rangle : I \rightarrow J$  where  $i_0, i_1 \dots, i_n$  represent a number of references to extended bigraph instances.

A composite bigraph will contain references to each of the component bigraphs. When a method is called on the signature, place graph or link graph of a composite bigraph, that method will in turn call the corresponding methods of each of the component bigraphs. For example, if a user wants to add a link type to a composite bigraph, it will in fact be added to each of the component bigraphs.

Because the structure of a composite bigraph is defined by parallel horizontal composition of its component bigraphs, links going from one component bigraph to another component bigraph will be defined as common outer names in each of the involved component bigraphs.

Internally, the individual component bigraphs function exactly as normal. Composition of composite bigraphs is not defined, and we will therefore not implement it.

## 5.3 Data access layer

We will implement a data access layer (DAL) and a data layer for a relational database and for XML. The persistence logic will be defined by a bigraph provider interface, and both data access layers will be an implementation of this. The interface will define two methods - one for loading and one for persisting.

Each data access layer instance will need some information in order to connect to the data source. For example, the relational database data access layer will need a database connection string, including the location of the database and a username and a password. In the case of the XML data access layer, the location of the XML document will be enough.

This is a very simple design that only provides a minimum of functionality. This is intentional as the main focus of this thesis is on the framework core.

This design forces us to load the whole bigraph whenever we want to access any part of the bigraph. Depending on how the user works with the bigraph this may not be very efficient, however, this design is dictated by the adoption of the base design principles of OWL.

Besides persistence of bigraph structures as defined by the framework, the data access layer will also handle export/import functionality. The main difference between normal persistence functionality and export/import functionality is that export/import functionality has to do with data formats that we do not define, e.g. OWL.

In our implementation we will, however, not implement export/import of any predefined formats. Instead we will implement export functionality to the XML format we have defined for the XML-based data layer as a proof of concept.

## 5.4 Data layer

The architecture of the framework provides room for several different data layer implementations, and as discussed earlier we shall provide two, namely a rela-

tional database implementation and a XML implementation.

There are no hard requirements for the data layer, except that it must be able to hold all the information held by the bigraph object model. If a proposed data layer cannot hold all this information, the data format of that data layer could instead be considered as an export format, as there are no requirements for those.

### 5.4.1 Relational database implementation

The relational database data layer implementation will closely resemble the object model, in that there will be a table for each class in the object model (except for interface implementations, see below) and each object instance will be stored as a row in the corresponding table. The properties of an object will be columns in that table. Collection type properties, however, will be stored in separate tables referencing the objects they belong to.

As in the XML format designed for the import/export functionality, the 16 byte unique identifiers will be employed as identifiers in references, and this implies that these identifiers will be the keys of the tables.

In some cases several classes implement the same interface, and therefore they are of the same type, e.g. the region, node and hole classes all implement the same interface. In such cases all instances will be stored in the same table and each instance will have a column indicating of what implementing type it is.

We will define as many restraints as possible to ensure data consistency. Therefore all references to other tables will have a corresponding foreign key constraint.

We will implement the relational database data layer implementation as a Microsoft SQL Server database, although it could be implemented as any relational database. We have chosen SQL Server because the .Net framework contains classes for easy access to SQL Server instances.

### 5.4.2 XML implementation

We will define the XML format ourselves. Because the XML format will be based on our object model of bigraphs, there will be a corresponding element for each object. The element names will be the same as the class names in the object model.

Object properties will be implemented as element attributes, where the value of the attribute is either a simple value, e.g. an integer or a string, or a reference to another attribute. This is excepting collection type properties which will be implemented as child elements with an element for each object in the collection.

All components will have unique identifiers as discussed on page 48. Where the object model employs class instance references for referencing components, the XML model will employ the unique identifiers.

For example, a node element will have a reference to its control. The value of the reference will be the value of the unique identifier on the control element, so that writing an XPath statement that locates the control element will be simple.

## 5.5 Summary

In this chapter we have provided an analysis of the architecture of the framework. We have defined the framework to be made up of three layers: the framework core, the data access layer and the data layer.

We have designed the framework core as two implementations: a base implementation strictly based on the defined bigraph theory and an extended implementation based on the base implementation and the extensions defined in chapter 4.

We have discussed how bigraphs can be modeled in an object oriented programming environment and in this context we have encountered two shortcomings of the theory: the lack of sharing of bigraph structures and the lack of a well defined way of defining generic transformations on bigraph structures.

Sharing of structures is a requirement for the taxonomy framework and it is a feature that is useful in many different applications. The workflow patterns use case, where workflows are built from predefined patterns, is an example of this. Here workflow patterns are shared to provide the separate steps of the workflows.

Currently composition and reaction rules are the only ways of altering the structure of bigraphs. This means that there is no generic way of e.g. adding a child node to a node. Composition requires two matching bigraphs and reaction rules require for the target bigraph to match the redex of the reaction rule.

The lack of return values in reaction rules reflects a more fundamental shortcoming: the lack of a way of querying bigraphs. The DOM-inspired methods that we have defined would be better reflected in the theory by a query language for bigraphs.

These shortcomings will be discussed in more detail in the perspectives section of the conclusion.

We have implemented a framework based on the analysis in this chapter. In the next chapter we will provide a description of the implemented framework.

## Chapter 6

# Implementation

In this chapter we will first discuss the overall structure of the framework, and then we will discuss how each component has been implemented.

### 6.1 Framework structure

As discussed in section 5.1, the framework consists of several layers:

- the framework core, which handles the business logic.
- the data access layer, which provides an interface for accessing data in various formats.
- and the data layer which defines how data is persisted.

The framework core is furthermore divided into two implementations:

- the base bigraph implementation, which is implemented according to the definitions in chapter 3.
- the extended bigraph implementation, which implements the extensions defined in section 4.6.

This structure is illustrated in figure 5.1 on page 47 and figure 5.2 on page 49.

### 6.2 Framework implementation

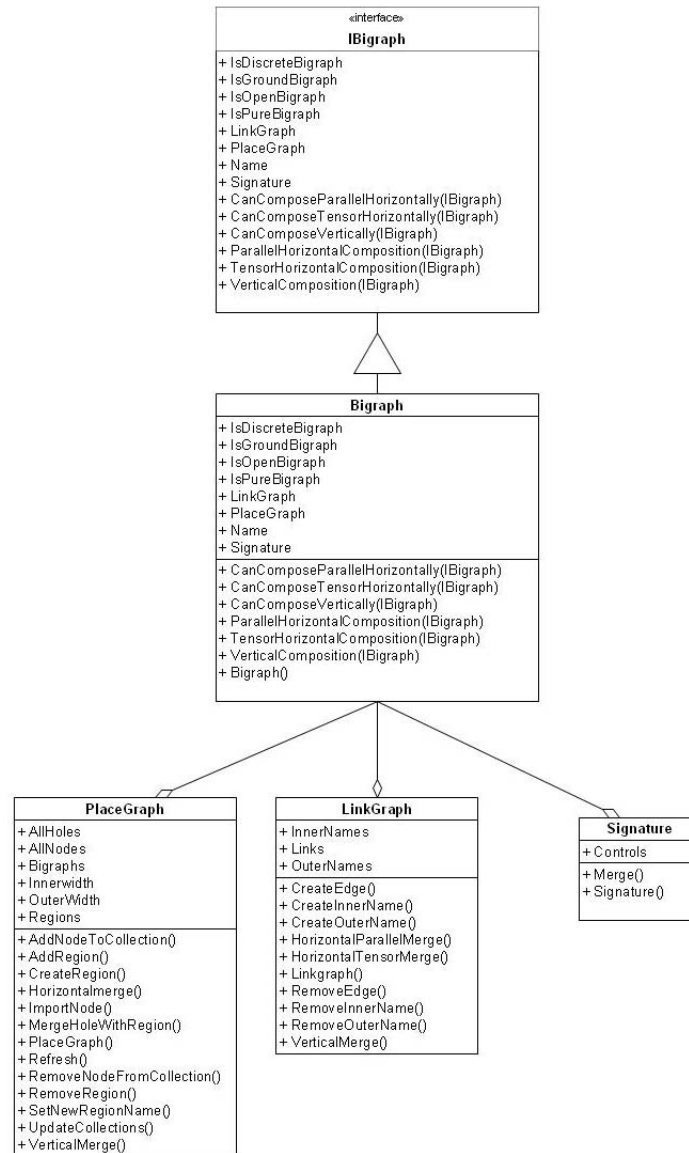
We will now provide a description of each of the components in the implementation. We will not discuss all details of the code. Rather we will discuss those sections of code that we think is of special interest.

While reading this section, the reader can refer to the source code which is provided in the appendix section of the thesis.

### 6.2.1 Base bigraph implementation

This implementation closely resembles the bigraph definitions provided in chapter 3. It is suitable for use by those who wish to use bigraphs as they are defined in the literature.

This implementation is designed so that it can easily be extended, which the next implementation is an example of.



diagrams/IBigraph.jpg

Figure 6.1: The IBigraph interface and major aggregation classes

## IBigraph

This interface defines the properties and methods that any bigraph implementation should have as a minimum.

The interface defines methods for composing vertically as well as horizontally. It also defines methods for checking if composition with another bigraph is at all possible.

The interface also defines methods for determining the type of the bigraph, i.e. if it is discrete, or ground etc.

Finally, the interface also provides access to the `Signature`, `PlaceGraph` and `LinkGraph` components.

## Bigraph

This is a minimal implementation of the `IBigraph` interface.

When an instance of the `Bigraph` class is created, it also creates its own `Signature`, `PlaceGraph` and `LinkGraph` properties. This is to ensure referential integrity between all the components.

The methods for determining the type of the bigraph simply check the conditions as described in section 3.2.1, and return `true` if the conditions are met.

The methods for determining if composition is possible take an `IBigraph` instance as an argument and then check the conditions as described in section 5.2.1. For example, when checking whether vertical composition is possible, the `InnerWidth` and `OuterWidth` and `InnerNames` and `OuterNames` of the instances will be compared, and if they match the method will return `true` otherwise `false`.

The composition methods first check whether composition is possible, and if it is not an exception is thrown. If composition is possible, the composition methods on the `PlaceGraph` and `LinkGraph` components are then called.

## PlaceGraph

A `PlaceGraph` instance has an `AllNodes` collection representing the set of nodes in the place graph definition. This is a readonly-collection, so it can only be used for iteration.

Additionally the `PlaceGraph` instance has `InnerWidth` and `OuterWidth` properties. When a `Hole` instance is added to or removed from the `PlaceGraph` instance the `InnerWidth` property is respectively increased or decreased by 1. The same applies for the `OuterWidth` instance and `Region` instances. This logic is handled by some of the methods defined on the `INode` interface.

A `PlaceGraph` instance has `AllHoles` and `Regions` collection properties for easy access to `Hole` and `Region` instances during composition.

The `PlaceGraph` class defines the instance method `CreateRegion` for creating a `Region` instance and adding it to the `Regions` collection property.

The `VerticalMerge` instance method handles vertical composition of two place graphs. First it is checked if the number of holes is equal to the number of regions. If this is the case the holes and regions are merged one by one.

The `HorizontalMerge` instance method handles both cases of vertical composition. This is done by simply moving the regions of one bigraph to the region collection of the other bigraph.

## LinkGraph

A `LinkGraph` instance has a `Links` collection which contains all the `ILink` instances belonging to that link graph. This is a readonly collection, so it can only be used for iteration.

The `LinkGraph` additionally has an `OuterNames` collection and an `InnerNames` collection containing respectively all the `OuterName` instances and all the `InnerName` instances of the link graph.

The `VerticalMerge` instance method handles vertical composition of two link graphs. This is basically done by merging the `OuterName` instances of the argument `LinkGraph` instance with the corresponding `InnerName` instances of the `LinkGraph` on which the method is called.

The `HorizontalParallelMerge` and `HorizontalTensorMerge` instance methods handle both cases of vertical composition. In the case of `HorizontalParallelMerge`, common `OuterName` instances are merged, and in the case of `HorizontalTensorMerge` an exception is thrown if there are common `OuterName` instances between the two `LinkGraph` instances.

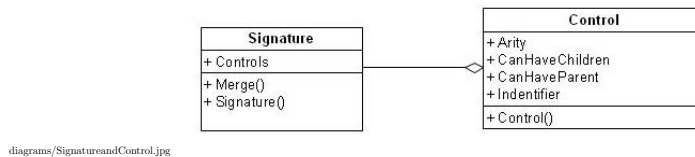


Figure 6.2: The Signature and Control class

## Signature

This class provides access to the `Control` instances of the `IBigraph` instance it belongs to. This is equivalent to the control set in bigraph theory.

The `Merge` method is used to merge two sets of `Control` instances during composition.

## Control

A `Control` instance has an `Arity` property. `Node` instances has a number of `Port` instances equal to the `Arity` property of the `Control` instance their `NodeControl` property points to.

## INode

The `INode` interface defines the interface that all implementations of elements in the place graph trees must adhere to. It is implemented by the `Region`, `Node` and `Hole` classes.

The DOM inspired properties and methods are defined on this interface. These are the `ChildNodes`, `HasChildNodes` and `ParentNode` properties and the `AppendChild`, `RemoveChild` and `ReplaceChild` methods.

The interface additionally defines the `DeepCopy` method which will copy an `INode` and all of its children.

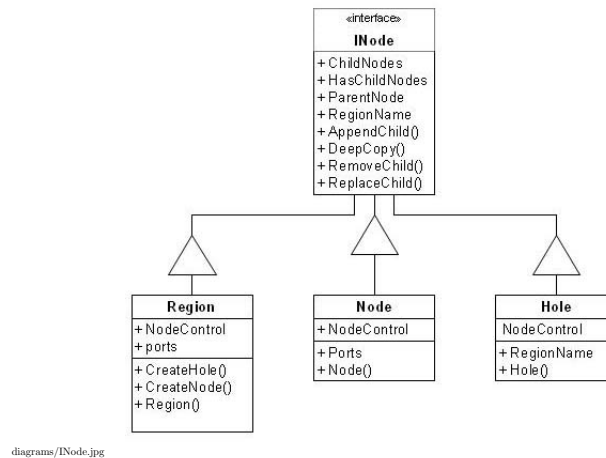


Figure 6.3: The INode interface and implementing classes

## Node

The **Node** class implements the **INode** interface.

The `HasChildNodes` property of a **Node** instance has the value `true` if the `ChildNodes` collection contains any **INode** instances. Otherwise it has the value `false`.

The `ParentNode` property contains a reference to the parent which is an **INode** instance.

The `AppendChild` instance method adds a **Node** or **Hole** instance to the `ChildNodes` collection of the **Node** instance and to the `AllNodes` collection of the **Bigraph** instance to which the **Node** belongs. If a **Hole** instance is added, the `InnerWidth` property of the **Bigraph** instance is increased by 1. If a **Region** instance is passed as the argument to this method, an exception will be thrown.

Similarly, the `RemoveChild` instance method removes a **Node** or **Hole** instance from the `ChildNodes` collection of the **Node** instance and from the `AllNodes` property of the **Bigraph** instance to which the **Node** instance belongs.

The `ReplaceChild` instance method first removes a child node and then appends a new node as described above.

A **Node** instance has a `NodeControl` property referencing a **Control** instance. The `Ports` collection of the **Node** instance contains a number of **Port** instances equal to the `Arity` property of the **Control** instance.

## Region

The **Region** class implements the **INode** interface.

The `ChildNodes` and `HasChildNodes` properties and the `AppendChild`, `RemoveChild` and `ReplaceChild` methods are defined similarly to the **Node** class.

The `ParentNode` property of **Region** instance will always be `null`. Instead the **PlaceGraph** instance will keep a reference to the **Region** instance.

The **Region** class additionally defines the methods `CreateHole` and `CreateNode`. These methods respectively create a **Hole** instance and an **Node** instance and adds them to the proper collection of the **Bigraph** instance.

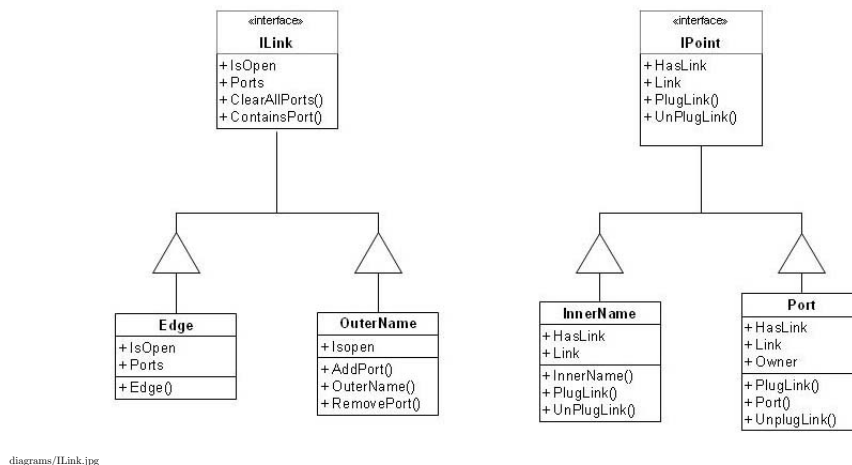
## Hole

The `Hole` class implements the `INode` interface.

The `ChildNodes` collection of a `Hole` instance will always be empty and consequently the `HasChildNodes` property instance has the value `false` and.

The `AppendChild`, `RemoveChild` and `ReplaceChild` methods of a `Hole` instance will always throw an exception if invoked.

The `ParentNode` property contains a reference to the parent which is an `INode` instance.



diagrams/ILink.jpg

Figure 6.4: The `ILink` and `IPoint` interfaces and implementing classes

## ILink

The `ILink` interface defines the interface that all link element implementations must implement. It is implemented by the `Edge` and `OuterName` classes.

It defines a `Ports` collection which should point to the `Port` instances to which an `ILink` instance is connected.

It also defines the methods `ClearAllPorts` and `ContainsPort` which respectively should clear the `Ports` collection and search through the `Port` collection for a specified `Port` instance.

## Edge

The `Edge` class implements the `ILink` interface.

The `Ports` property and the `ClearAllPorts` and `ContainsPort` instance methods are implemented as described above.

## OuterName

The `OuterName` class implements the `ILink` interface.

The properties and methods of the `OuterName` class are implemented as described for the `Edge` class.

## IPoint

The `IPoint` interface defines the interface that all point element implementations must implement. It is implemented by the `Port` and `InnerName` classes.

The `IPoint` interface defines a `HasLink` property which should be set to the boolean value `true` only if the `IPoint` instance is connected to an `ILink` instance.

The interface also defines the methods `PlugLink` and `UnPlugLink` for respectively plugging an `ILink` instance to the `IPoint` instance and unplugging an `ILink` instance from the `IPoint` instance.

## Port

The `Port` class implements the `IPoint` interface.

The `Owner` property of a `Port` instance contains a reference to the `Node` instance which the `Port` instance belongs to.

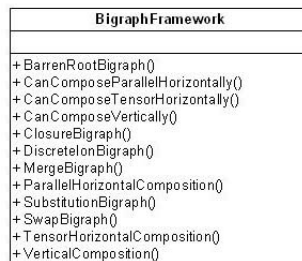
If the `Port` instance is connected to an `ILink` instance, the `Link` property contains a reference to that `ILink` instance.

The `PlugLink` and `UnPlugLink` instance methods respectively sets the `Link` property to an `ILink` instance or to `null`. If `PlugLink` is called when the value of `Link` is not `null`, an exception is thrown.

## InnerName

The `InnerName` class implements the `IPoint` interface.

The properties and methods defined by the `IPoint` interface are implemented as described for the `Port` class.



diagrams/BigraphFramework.jpg

Figure 6.5: The `BigraphFramework` class

## BigraphFramework

The `BigraphFramework` class defines factory methods for building block bigraphs. There is one method for each of the six building blocks.

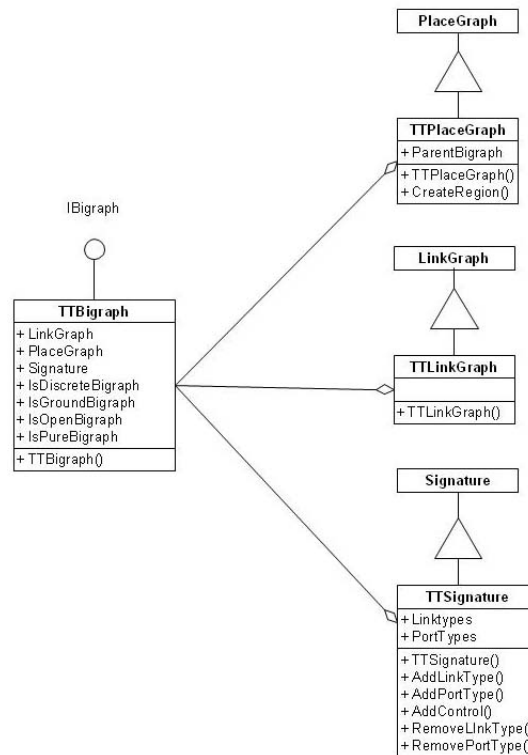
Some of these methods take parameters indicating the exact structure of the building block to be instantiated. For example, the `MergeBigraph` method, which instantiates a merge building block, takes a number indicating the number of holes to instantiate the building block with.

The `BigraphFramework` class also defines methods for composition. Each method takes two `IBigraph` instances as arguments and then calls the corresponding composition method on the first `IBigraph` instance and supplies the second `IBigraph` instance as the sole argument.

## 6.2.2 Extended bigraph implementation

In this section we will describe the extended bigraph implementation. As previously discussed the extended bigraph implementation is built on top of the base bigraph implementation. We will therefore only describe the extension features of these classes.

The names of all classes in the extended bigraph implementation are prefixed with “TT-”. This is because the implementation is designed specifically for use with the taxonomy framework. The old Taxonomy Tool is for historical reasons nicknamed “TT” within the company.



diagrams/TTBigraph.jpg

Figure 6.6: The TTBigraph class and major aggregation classes

### TTBigraph

The `TTBigraph` class implements the `IBigraph` interface and extends the `Bigraph` class from the base implementation.

This class does not add any functionality but instead of referencing a `Signature` class instance, a `PlaceGraph` class instance and a `LinkGraph` class instance

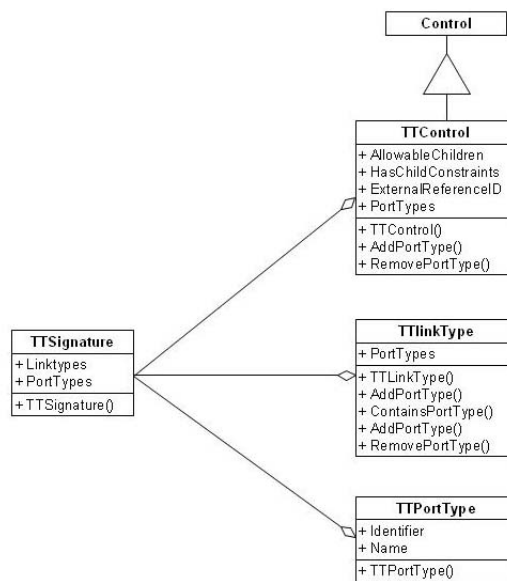
it references a `TTSignature` class instance, a `TTPlaceGraph` instance and a `TTLinkGraph` instance.

### TTPlaceGraph

The `TTPlaceGraph` extends the `PlaceGraph` class from the base implementation. Like the `TTBigraph` class it does not add any properties or methods.

### TTLinkGraph

The `TTLinkGraph` class extends the `LinkGraph` class from the base implementation. It does not add any properties or methods.



diagrams/TTSignature.jpg

Figure 6.7: The `TTSignature` class and aggregation classes

### TTSignature

The `TTSignature` class extends the `signature` class from the base implementation.

A `TTSignature` instance has a `TTControl` collection, a `TTLinkType` collection and a `TTPortType` collection. All `TTControl`, `TTLinkType` and `TTPortType` instances that are referenced within the bigraph to which the `TTSignature` instance belongs will be included in those collections.

The `TTSignature` also defines instance methods for adding and removing those components.

### TTControl

The `TTControl` class extends the `Control` class from the base implementation.

A `TTControl` instance has an `AllowableChildren` property which is a collection of `TTControl` instances. The children nodes of a `TTNode` instance must be either `TTHole` instances or `TTNode` instances whose `NodeControl` property is set to one of the `TTControl` instances in the `AllowableChildren` collection property of the parent `TTNode` instance.

The `PortTypes` property is a `TTPortType` collection containing a number of `TTPortType` instance references equal to the value of the `Arity` property of the `TTControl`. This property defines exactly how many and what types of `TTPort` instances `TTNode` instances referencing the `TTControl` reference must have.

The `TTControl` class also defines instance methods for adding and removing `PortType` instances to or from the `PortTypes` collection property.

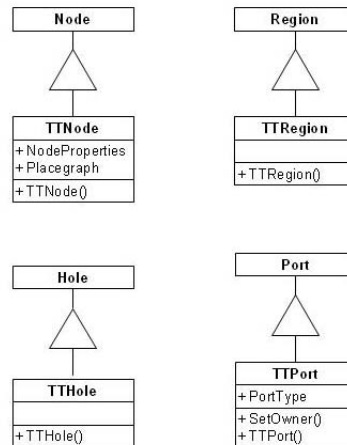
### TTLinkType

A `TTLinkType` instance has a `PortTypes` collection property. A `TTEdge` or `TTOuterName` can only be connected to a `TTPort` instance if the `TTPortType` of that `TTPort` instance is included in the `PortTypes` collection property of the `TTLinkType` to which the `LinkType` property of the `TTEdge` or `TTOuterName` points.

The `TTLinkType` class defines methods for adding or removing `PortType` instances to or from the `PortTypes` collection property and for checking whether or not the collection contains a specific `PortType` instance.

### TTPortType

A `TTPortType` instance has no special properties or methods, but it is used when determining if a `TTPort` instance can be connected to a `TTEdge` or `TTOuterName` instance.



diagrams/TTRegion.jpg

Figure 6.8: The extension classes implementing the `INode` interface and the `TTPort` class

### TTNode

The `TTNode` class implements the `INode` interface and extends the `Node` class from the base implementation. The `TTNode` class does not add any properties or methods to the base class.

When a `TTNode` instance is created the `Ports` collection property of that instance is instantiated with a number of `TTPort` instances as specified by the `PortTypes` property of the `TTControl` to which the `Control` property of the `TTNode` instance points.

When the `AppendChild` or `ReplaceChild` methods of some `TTNode` instance is called it is checked whether the operation is legal. If the child to be added is a `Hole` instance the operation is legal. If the child to be added is a `TTNode` instance whose `TTControl` instance is referenced in the `TTNode` instance's `AllowableChildren` collection property the operation is legal. Otherwise the operation is illegal and an exception is thrown.

The `NodeProperties` collection is a collection of key/value string pairs.

### TTRegion

The `TTRegion` class extends the `Region` class from the base implementation. It does not add any properties or methods.

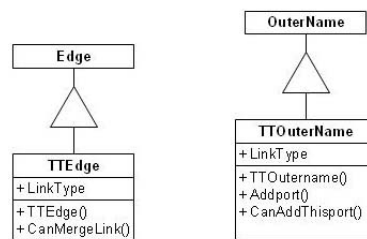
### TTHole

The `TTHole` class extends the `Hole` class from the base implementation. It does not add any properties or methods.

### TTPort

The `TTPort` class extends the `Port` class from the base implementation.

A `TTPort` instance has a `PortType` property pointing to a `TTPortType` instance. This is used for checking if the `TTPort` instance can be connected to a `TTEdge` or `TTOuterName` instance.



diagrams/TTOuterName.jpg

Figure 6.9: The extension classes implementing the `ILink` interface

### TTEdge

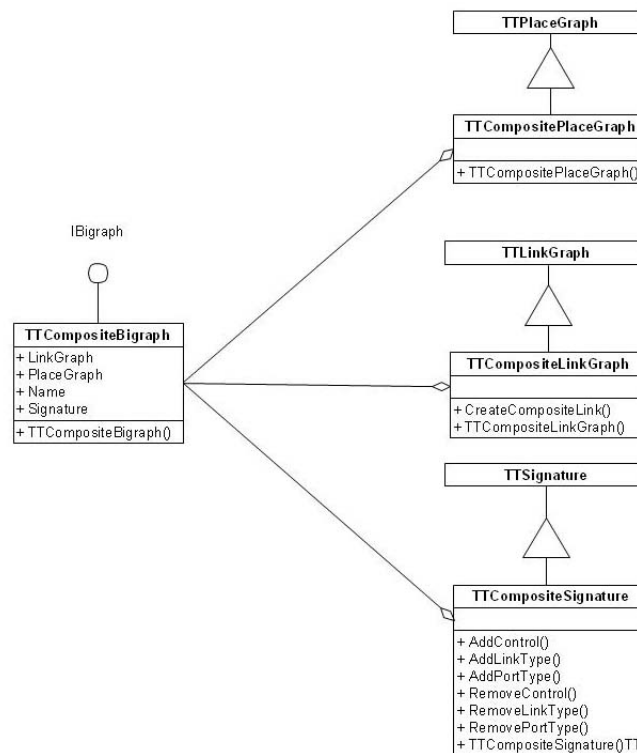
The `TTEdge` implements the `ILink` interface and extends the `Edge` class from the base implementation.

A `TTEdge` instance has a `LinkType` property pointing to a `TTLinkType` instance.

### TTOuterName

The `TTOuterName` class implements the `ILink` interface and extends the `OuterName` class from the base implementation.

A `TTOuterName` instance has a `LinkType` property pointing to a `TTLinkType` instance.



diagrams/TTCompositeBigraph.jpg

Figure 6.10: The `TTCompositeBigraph` class and aggregate classes

### TTCompositeBigraph

The `TTCompositeBigraph` class does not implement the `IBigraph` nor does it extend the `Bigraph` or the `TTBigraph` classes. Consequently, `TTCompositeBigraph` instances cannot be applied as the argument to any of the composition methods of the classes that implement the `IBigraph` interface.

A `TTCompositeBigraph` instance has `Signature` property pointing to a `TTCompositeSignature` instance, a `PlaceGraph` instance pointing to a `TTCompositePlaceGraph` instance and a `PlaceGraph` property pointing to a `TTCompositePlaceGraph` instance.

Internally, a `TTCompositeBigraph` instance has a collection of component `TTBigraph` instances. It also has methods for adding instances to the collection

and removing them from the collection.

### **TTCompositeSignature**

The `TTCompositeSignature` class extends the `TTSignature` class.

When the `Controls` property is accessed, a `TControl` instance collection containing the content of the `Controls` property of the `TTSignature` instance of each of the component `TBigraph` instances, but with duplicates removed, is returned.

The `LinkTypes` and `PortTypes` properties are implemented in the same way as the `Controls` property, except they return respectively a `TTLinkType` and a `TTPortType` collection.

### **TTCompositePlaceGraph**

The `TTCompositePlaceGraph` class extends the `TTPlaceGraph` class.

The `AllHoles`, `AllNodes` and `Regions` collection properties function like the `Controls` property of the `TTCompositeSignature` class. When accessed a collection containing the sum of the respective properties on each component `TBigraph` instance, with duplicates removed, is returned.

Similarly, the `OuterWidth` and `InnerWidth` properties return the sum of respectively the `OuterWidth` or `InnerWidth` properties of all component `TBigraph` instances.

### **TTCompositeLinkGraph**

The `TTCompositeLinkGraph` class extends the `TTLinkGraph` class.

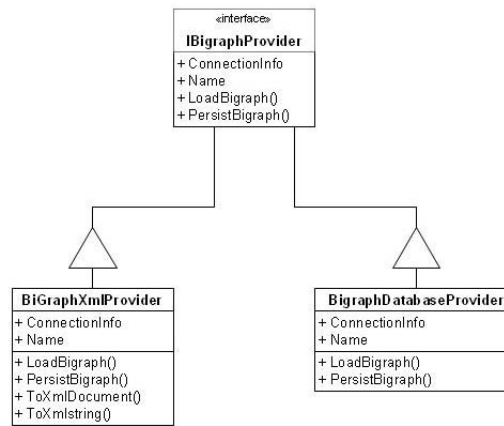
As with the properties of the `TTCompositeSignature` and `TTCompositePlaceGraph` classes, the properties of the `TTCompositeLinkGraph` are built from the respective properties of the component `TBigraph` instances.

A `TTCompositeLinkGraph` instance has a `CreateCompositeLink` method. When called with an `IPoint` instance and an `ILink` instance from two different component `TBigraph` instances it adds a `TTOuterName` instance to each of the `OuterNames` collection properties of the two `TBigraph` instances involved. The two `TTOuterName` instances will have the same name, indicating that they are connected. If the `IPoint` instance and the `ILink` instance is from the same bigraph an exception will be thrown.

Note that, although the implementation is different, the effect is the same as storing a link as outer names in the component bigraphs and inner names in the context bigraph. Since `TTCompositeBigraph` does not implement the `IBigraph` interface this is, however, not possible.

## **6.2.3 Data access layer**

We have implemented two data access layer implementations: one for extended bigraphs stored in relational databases and one for extended bigraphs stored in XML documents.



diagrams/IBigraphProvider.jpg

Figure 6.11: The IBigraphProvider interface and implementing classes

### IBigraphProvider

The **IBigraphProvider** interface defines the properties and methods that all data layer access implementations must implement.

The **ConnectionInfo** string property will hold any information necessary for accessing the data source. The format of this format should be XML-based.

The **LoadBigraph** and **PersistBigraph** are methods for respectively loading a bigraph from the data source and persisting a bigraph to the data source. When each of these methods are called the whole bigraph is either loaded or persisted.

### BiGraphXMLProvider

The **BiGraphXMLProvider** class implements the **IBigraphProvider** interface.

The **ConnectionInfo** property contains a file path which points to an XML document.

The **LoadBigraph** parses an XML document and, while doing so, builds the **TTBigraph** instance. The **PersistBigraph** parses a **TTBigraph** instance and while doing so builds an XML document which is finally stored. The XML format is as described in the following section.

The **ToXmlDocument** method exports a **TTBigraph** instance to an XML document based on the same XML format as above described methods.

### BiGraphDatabaseProvider

The **BiGraphDatabaseProvider** class implements the **IBigraphProvider** interface.

The **ConnectionInfo** property contains a database connection string which includes the location of the database and a username and password pair.

The **LoadBigraph** and **PersistBigraph** methods respectively load a **TTBigraph** instance from and persists a **TTBigraph** instance to a database. The format of the database is as described in the next section.

## 6.2.4 Data layer

We have implemented two data layers based on the analysis in section 5.4.

### Relational database

The relational database data layer is implemented as an MS SQL Server database. All identifiers in the database are of data type `uniqueidentifier` which is a 16 byte randomly generated string. All relations are defined with foreign key constraints. A diagram illustrating the database table structure can be found in the appendix.

The `Bigraphs` table contains all bigraph instances. Only the unique identifier and name of each bigraph instance is stored in this table. Elements in other tables, e.g. the `TreeElements` or `Links` tables, reference the bigraph they belong to.

The `TreeElements` table contains all the elements which constitute the place graphs of the bigraphs, i.e. regions, nodes and holes. The `TreeElementTypes` table contains a row for each defined element type, and each row in the `TreeElements` table references one of these types, indicating of what type the tree element in that row is.

Each tree element in the `TreeElements` table is related to one control in the `Controls` table and zero or more ports in the `Ports` table. Each tree element can furthermore be related to another row in the `TreeElements` table indicating a parent-child relationship.

Each row in the `Controls` table represents a control instance. The `ChildControls` table defines what controls a control can have as children and the `ControlToPortType` table defines what ports nodes of each control must have.

Each row in the `Controls` table may also have a reference to the `ExternalRefInfo` table. The `ExternalRefInfo` table contains information about how to access external data. The `SourceInfo` column contains the necessary parameters to access external data in a predefined XML format. Each row in the `ExternalRefInfo` table references a row in the `ExternalRefTypes` table indicating of what type the external reference is, e.g. database or XML, and consequently how the data should be accessed with aforementioned parameters.

The `Ports` table contains all the port instances that belong to node instances. Each row in the `Ports` table has a reference to a row in the `PortTypes` table indicating the type of the port. A row may also have a reference to a row in the `Links` table indicating that it is connected to that link instance.

The `Links` table contains all link instances. Each link has a 1-bit column indicating whether it is an edge or an outer name. Each row in the `links` table has a reference to a row in the `LinkTypes` table indicating the type of the link. The `LinkTypeToPortType` table defines what types of links can be connected to what types of ports.

The `InnerNames` table contains all inner name instances. each row in the `InnerNames` table may reference a row in the `Links` table indicating a connection.

### XML

In the XML structure aggregation of objects is generally indicated by hierarchical relations, i.e. an element contained within another element. As in

the relational database implementation all elements have a 16 byte randomly generated unique identifier. In this description we will only mention element attributes that are of particular interest.

The root element of the bigraph XML structure is a **Bigraph** element. The **Bigraph** element contains a **Signature** element, a **PlaceGraph** element and a **LinkGraph** element.

Since this implementation corresponds to the extended bigraph implementation the **Signature** element contains both a **ControlCollection** element, a **LinkTypeCollection** element and a **PortTypeCollection** element. Correspondingly, each of these contain a number of **Control**, **LinkType** or **PortType** elements.

The **PlaceGraph** contains a **RegionCollection** element which contains a number of **Region** elements. Each **Region** element may contain a number of **Node** elements and **Hole** elements.

Each **Node** element contains a **PortCollection** element which contains a number of **Port** elements. Each **Port** element has a **linkReference** attribute which may contain the identifier value of an **Edge** or and **OuterName** element.

Each **Node** element may furthermore have a **ChildNodes** element containing the children nodes of that node.

The **LinkGraph** element contains an **OuterNameCollection**, an **InnerNameCollection** and an **EdgeCollection**.

The **OuterNameCollection** and **InnerNameCollection** contain respectively a number of **Outername** elements and a number of **InnerName** elements. The **EdgeCollection** contains a number **Edge** elements.

Each **OuterName** and **Edge** element contains a **PortCollection** element which will contain a **LinkPort** reference for each port the link is connected to.

### 6.3 Summary

In this chapter we have presented how we have implemented the implementation of the framework based on the previous analysis chapters. We will now move on to presenting a short guide on how to use the framework for programming with bigraph-based data structures.

# Chapter 7

## User Guide

In this chapter we will present a guide on how to use the framework that we have implemented in this project. We will not go into all details, but will rather present a brief introduction so that users can get started quickly.

We will be basing this user guide on the forms and laws example presented earlier and we will be using the extended bigraph implementation.

### 7.1 Creating a bigraph

Creating a bigraph is done by simply calling the constructor of the `TTBigraph` class. The constructor takes a string as the only argument, this will be the name of the bigraph.

We will need the `TTSignature`, `TTPlaceGraph` and `LinkGraph` instances later so we will store references to these now.

```
//Step 1. Create a new instance of TTBigraph
// and get the references to its components.

// Create a new instance of Bigraph by passing the name for the Bigraph.
TTBigraph tt_bigraph_host_forms = new TTBigraph("Forms");

// Get the reference to TTSignature.
TTSignature tt_signature = tt_bigraph_host_forms.Signature;

// Get the reference to TTPlaceGraph.
TTPlaceGraph tt_place_graph = tt_bigraph_host_forms.PlaceGraph;

// Get the reference to TTLinkGraph.
TTLinkGraph tt_link_graph = tt_bigraph_host_forms.LinkGraph;
```

Since we want to be able to create links between nodes in the bigraph we have to create the port types and link types and add them to the signature. First we create the port types.

```

// Step 2. Create the TTPortType instances and add them to TTSignature.

// create a new instance of TTPortType Link between Form to Authority.
TTPortType tt_port_type_form_to_authority =
    new TTPortType("FormToAuthorityConnectionPort");

// Add the instance of TTPortType to the TTSignature.
tt_signature.AddPortType(tt_port_type_form_to_authority);

// Create a new instance of PortType for Link between Form To Form.
TTPortType tt_port_type_form_to_form = new TTPortType("FormToFormConnectionPort");

tt_signature.AddPortType(tt_port_type_form_to_form);

// Create a new instance of TTPortType for Link between Authority to Form.
TTPortType tt_port_type_authority = new TTPortType("AuthorityToFormConnectionPort");

tt_signature.AddPortType(tt_port_type_authority);

TTPortType tt_port_type_Law_to_Forms = new TTPortType("LawsToFormsConnectionPort");

// Add the instance of TTPortType to the TTSignature.
tt_signature.AddPortType(tt_port_type_Law_to_Forms);

TTPortType tt_port_type_Forms_to_Law = new TTPortType("FormsToLawsConnectionPort");

// Add the instance of TTPortType to the TTSignature.
tt_signature.AddPortType(tt_port_type_Forms_to_Law);

```

Then we create the link types.

```

// Step 3. Create the TTLinkType instances and add them to TTSignature.

// Create a new instance of TTLinkType for defining type of Link
// between the Authority and Forms. TTLinkType
tt_link_type_authority_to_form = new TTLinkType("AuthorityToForm");

// Add the Authority Port type to participate in the Link at the most once.
// Here we define the Occurrence index to 1 which means that Authority can
// participate at the most once in the link.
tt_link_type_authority_to_form.AddPortType(tt_port_type_authority, "1");

// Add the Form Port type to this Linktype.
// Here we skip the Occurrence index,
// so the Form Port can participate unlimited number of times.
tt_link_type_authority_to_form.AddPortType(tt_port_type_form_to_authority);

```

```

// add the linkType to TTSignature
tt_signature.AddLinkType(tt_link_type_authority_to_form);

TTLinkType tt_link_type_form_to_form = new TTLinkType("Related
Forms");

// The definition of the LinkType permits Form ports to participate in the link
// any number of time..
tt_link_type_form_to_form.AddPortType(tt_port_type_form_to_form, "*");

// add the linkType to TTSignature
tt_signature.AddLinkType(tt_link_type_form_to_form);

// Create LinkTypes.
TTLinkType tt_link_type_Laws_to_form = new TTLinkType("LawsToForms");

// The definition of the LinkType permits Form ports to participate in the link
// any number of time..
tt_link_type_Laws_to_form.AddPortType(tt_port_type_Law_to_Forms);

tt_link_type_Laws_to_form.AddPortType(tt_port_type_Forms_to_Law);

// add the linkType to TTSignature
tt_signature.AddLinkType(tt_link_type_Laws_to_form);

```

We also need to create the controls that we will later assign to the nodes. We define a control for the authority nodes and a control for the form nodes.

```

// Step 4 Creation of TTControls.

// create a new instance of TTControl for Authority..
TTControl tt_control_authority = new TTControl("Authority");

// Add PortType so that all authority nodes will have this port by default.
tt_control_authority.AddPortType(tt_port_type_authority);

// Assign the can have children property to true
// so that it can contain the children.
tt_control_authority.CanHaveChildren = true;

// add the control to the Signature.
tt_signature.AddControl(tt_control_authority);

TTControl tt_control_form = new TTControl("Form");

// add port types.
tt_control_form.AddPortType(tt_port_type_form_to_authority);

```

```

tt_control_form.AddPortType(tt_port_type_form_to_form);

tt_control_form.AddPortType(tt_port_type_Forms_to_Law);

// Add the control to the TTSignature.
tt_signature.AddControl(tt_control_form);

// Add a child constraint on Authority control, so that only
// Form controls can be added to the Authority.

tt_control_authority.HasChildConstraints = true;

// add the Forms control to Allowable children for the Authority..
tt_control_authority.AllowableChildren.Add(tt_control_form);

```

We then proceed to create the regions, holes and nodes of the place graph.

```

// Step 5 Creation of Region and Nodes..

TTRegion tt_region = tt_place_graph.CreateRegion("Region 0");

// create a new Forms Authority..
TTNode tt_authority_node =
    tt_region.CreateNode(tt_control_authority, "Toldskat");

// Add the authority node to the Region..
tt_region.AppendChild(tt_authority_node);

// Create 2 Forms and add it to the Authority node.
TTNode tt_node_form1 = tt_region.CreateNode(tt_control_form, "Form No.1");

TTNode tt_node_form2 = tt_region.CreateNode(tt_control_form, "Form No.2");

// Create a Hole in the under Authority Node
TTHole tt_hole = tt_region.CreateHole();

// Add the Form node and and hole to the AuthorityNode.
tt_authority_node.AppendChild(tt_node_form1);

tt_authority_node.AppendChild(tt_node_form2);

tt_authority_node.AppendChild(tt_hole);

```

We now have a full bigraph. We have yet to define any links between the nodes however.

## 7.2 Linking nodes

We will now begin creating links and connecting the nodes to these links. We will also create two outer names and an inner name.

```
// 1. Create link between Form and Authority..

// Create a ports collection..
PortCollection port_collection = new PortCollection();

// TTEdges
// Get the reference of the First Port on the authority
// which is connecting port to the Form.
port_collection.Add(tt_authority_node.Ports[0]);

// Get the reference of the First Port on the Forms nodes
// which are connecting port to the authority.
port_collection.Add(tt_node_form1.Ports[0]);

port_collection.Add(tt_node_form2.Ports[0]);

// create new TTEdge connecting the these ports .
TTEdge tt_edge1 =
tt_bigraph_host_forms.LinkGraph.CreateEdge(tt_link_type_authority_to_form,
      "AuthorityToForm",port_collection);

// create a new instance of PortCollection..
port_collection = new PortCollection();

// Get the reference to the II Port of the on the Forms
// an create a Link between them.
port_collection.Add(tt_node_form1.Ports[1]);

port_collection.Add(tt_node_form2.Ports[1]);

// Create a new instance of TTEdge connecting the Ports on the Forms.
TTEdge tt_edge_2 =
tt_bigraph_host_forms.LinkGraph.CreateEdge(tt_link_type_form_to_form,
      "Related forms", port_collection);

//TTOuterNames

port_collection = new PortCollection();

port_collection.Add(tt_node_form2.Ports[2]);

TTOuterName tt_outer_name_0 =
tt_link_graph.CreateOuterName(tt_link_type_Laws_to_form,
      "Form2 To Law#346", port_collection);
```

```

port_collection = new PortCollection();

port_collection.Add(tt_node_form1.Ports[2]);

TTOuterName tt_outer_name_1 =
tt_link_graph.CreateOuterName(tt_link_type_Laws_to_form,
"Form1ToLaw#786",port_collection);

// InnerNames
// Create InnerName connected to tt_edge1 ...
InnerName inner_name_1 =
tt_bigraph_host_forms.LinkGraph.CreateInnerName("FormsLink",
tt_edge1);

```

### 7.3 Persistence

We will now persist the bigraph created so far in an XML document.

```

// Bigraph Persistence

// Create a new instance of BigraphXmlProvider for saving the Bigraph.
BigraphXmlProvider bigraph_xml_provider = new BigraphXmlProvider();

// Assign the XML file name.
bigraph_xml_provider.ConnectionInfo = "../..Forms_host_bigraph.xml";

bigraph_xml_provider.PersistBigraph(tt_bigraph_host_forms);

```

### 7.4 Composition

We will now show how composition is performed. We will assume the existence of a second bigraph `tt_guest_bigraph` whose outer face matches the inner face of the bigraph created so far.

```

// Loading guest Bigraph from XML file.

// Create a new instance of BigraphXmlProvider for saving the Bigraph.
//BigraphXmlProvider bigraph_xml_provider = new BigraphXmlProvider();

// Assign the XML file name.
//bigraph_xml_provider.ConnectionInfo = "../..Forms_host_bigraph.xml";

TTBigraph tt_guest_bigraph = bigraph_xml_provider.LoadBigraph(
    "../..Forms_guest_bigraph.xml");

```

```

// Bigraph Composition

bool can_compose = tt_bigraph_host_forms.CanComposeVertically(
    tt_guest_bigraph);

// Here tt_guest_bigraph will be composed into the tt_bigraph_host_forms.
if (can_compose)
{
    tt_bigraph_host_forms.VerticalComposition(tt_guest_bigraph);
}

```

## 7.5 Composite bigraph

Finally we will show how to create a composite bigraph. We will assume the existence of a laws bigraph `tt_bigraph_laws` which will be the second component bigraph along with the bigraph created above. These two bigraph will be the component bigraphs of the composite bigraph.

```

// create a new instance of TTCompositeBigraph..
tt_composite_bigraph = new TTCompositeBigraph("FormsAndLaws");

// Add forms bigraph..
tt_composite_bigraph.AddBigraph(tt_bigraph_forms.Name, tt_bigraph_forms);

// Add Laws Bigraph..
tt_composite_bigraph.AddBigraph(tt_bigraph_laws.Name, tt_bigraph_laws);

// Composite Link

// Create a new Form "Form 007" in

// Get the TTControl..
TTControl tt_control_form = (TTControl)tt_composite_bigraph.GetBigraph(
    "Forms").Signature.Controls["Form"];

TTRegion tt_region_forms =(TTRegion) tt_composite_bigraph.GetBigraph(
    "Forms").PlaceGraph.Regions[0];

TTNode tt_node_from_007 = tt_region_forms.CreateNode(tt_control_form,
    "Form 007");

// Append the new ly create form node to the Authority Node.
tt_region_forms.ChildNodes[0].AppendChild(tt_node_from_007);

// Create a new Law "Law #999"..

```

```
// Get the TTControl for Law node..
TTControl tt_control_law = (TTControl)tt_composite_bigraph.GetBigraph(
    "Laws").Signature.Controls["Law"];

TTRegion tt_region_law = (TTRegion)tt_composite_bigraph.GetBigraph(
    "Laws").PlaceGraph.Regions[0];

TTNode tt_node_law_999 = tt_region_law.CreateNode(tt_control_law,
    "Law #999");

// Append it to the authority node.
tt_region_law.AppendChild(tt_node_law_999);

// create a Composite link between Form Node "Form 007"
// and Law node "Law #999". PortCollection port_collection = new
PortCollection();

port_collection.Add(tt_node_from_007.Ports[2]);

port_collection.Add(tt_node_law_999.Ports[0]);

// Get the LinkType..
TTLinkType tt_linl_type_Forms_to_Laws = tt_composite_bigraph.GetBigraph(
    "Forms").Signature.LinkTypes["LawsToForms"];

// Create compositeLink..
TTCompositeLinkGraph tt_composite_link_graph =
    (TTCompositeLinkGraph)tt_composite_bigraph.LinkGraph;

TTOuterName[] tt_composite_outer_names =
tt_composite_link_graph.CreateCompositeLink(tt_linl_type_Forms_to_Laws,
"CompositeLink Form 007 to Law #999", port_collection);
```

## 7.6 Summary

In this chapter we have presented a brief guide including code samples to help users get started with the framework. We will now proceed to describe how we have tested the framework.

# Chapter 8

## Test

In this chapter we will describe the tests which we have defined and performed in order to ensure the quality of the code.

We will not define tests covering the base bigraph implementation. As the extended bigraph implementation is the main focus of this thesis, we will concentrate on testing that. Also, as the extended bigraph implementation is an extension of the base bigraph implementation, parts of the base bigraph implementation will be covered by this test. If we were to test the base bigraph implementation, the test would be very similar to the test described in this chapter.

### 8.1 Test method

As testing is not the main purpose of this thesis, we have decided to be content with a test that simply ensures that the application works according to the design.

We will define a number of unit tests. We will not define test cases for all classes in the code as a proper unit test should have, but rather we will define use cases for all code entrances. We predict that at least 90% of the code will be covered by such a test.

We will output the resulting bigraph of each test as XML and manually check that the result is as expected. We will list the result of all tests at the end of this chapter and we will provide the output of all tests in an appendix.

If any test results in an unexpected result, we will change the code so that part of the code works as expected. This will be done in an iterative process, so that in the end all tested functionality works as we expect it to.

### 8.2 Test cases

#### 8.2.1 Builder methods

These test cases will test if the various builder methods work correctly.

### T1.1 Build a bigraph

Create an instance of the `TBigraph` class. Check that it correctly builds a `TTSignature` instance, a `TTPlaceGraph` instance and a `TTLinkGraph` instance as properties on the `TBigraph` instance.

### T1.2 Add a port type, link type and a control

Create a `TTPortType` instance and pass it as the argument to the `AddPortType` method on the `TTSignature` instance.

Create a `TTLinkType` instance and pass it as the argument to the `AddLinkType` method on the `TTSignature` instance.

Create a `TTControl` instance and pass it as the argument to the `AddControl` method on the `TTSignature` instance. Add the `TTPortType` instance to the `PortTypes` collection of the `TTControl` instance.

Check that both instances have been added to their respective collections.

### T1.3 Add a region

Call the `CreateRegion` on the `PlaceGraph` instance. Check that a `Region` instance has been added to the `Regions` collection of the `TTPlaceGraph`.

### T1.4 Add nodes

Call the `CreateNode` on the `Region` instance. Specify the `TTControl` instance as the control of the created node. Ensure that the method returns a `TTNode` instance.

Call the `AppendChild` method of the `Region` instance with the `TTNode` as an argument. Check that the `TTNode` instance is now the child of the `Region` instance.

Repeat this with a `TTHole` instance instead of the `TTNode` instance.

### T1.5 Add inner names and outer names

Call the `CreateEdge` method on the `TTLinkGraph` instance. Pass the inner name and the `TTPort` instances of the two `TTNode` instances as arguments to this method. Check that the inner name and the two nodes are connected to the same `TTEdge` instance.

### T1.6 Add outer name

Call the `CreateOuterName` method on the `TTLinkGraph` instance. Check that the inner name has been added correctly.

### T1.7 Add inner name

Call the `AddInnerName` method on the `TTLinkGraph` instance. Check that the inner name has been added correctly.

## 8.2.2 Composition

These test cases all use two `TTBigraph` instances. The previously created instance we will call *A*. The other `TTBigraph` instance we will call *B* and will have two regions and a name in the outer face equal to the name in the inner face of the previously created bigraph. One of the regions must contain a node that is connected to the outer name.

These bigraphs should be recreated before each test and discarded after that test is completed.

### T2.1 Vertical composition

Call the `VerticalMerge` on *A* and pass *B* as the argument. Check that the composition succeeds, and that the node from *B* is added to the edge in *A*.

Repeat the operation again first with the outer name removed and then one of the regions removed. Check that the operation fails both times.

### T2.2 Tensor horizontal composition

Call the `TensorHorizontalComposition` method on *A* and pass *B* as the argument. Check that the composition fails because the bigraphs share the same outer name. Remove the outer name from *B* and repeat the operation. Check that the composition succeeds and that *A* now has 3 regions.

### T2.3 Parallel horizontal composition

Call the `ParallelHorizontalComposition` method on *A* and pass *B* as the argument. Check that the composition succeeds and that *A* now has 3 regions but still only 1 outer name.

## 8.2.3 Composite bigraphs

The following tests will be using the same bigraph instances *A* and *B* as those used in section 8.2.2.

### T3.1 Create a composite bigraph

Create a `TTCompositeBigraph` instance. Check that it correctly builds a `TTCompositeSignature` instance, a `TTCompositePlaceGraph` instance and a `TTCompositeLinkGraph` instance as properties on the `TTCompositeBigraph` instance.

Call the `AddBigraph` method twice with respectively *A* then *B* as the argument. Check that the `TTCompositePlaceGraph` has exactly 3 regions.

### T3.2 Create a link between the component bigraphs

Call `CreateCompositeLink` on the `TTCompositeLinkGraph` instance and pass it a node from *A* and a node from *B* as arguments. Check that the link is created and is connected to the two nodes. Check that one outer name has been added to the outer face of respectively *A* and *B* and that each outer name is connected to the specified nodes.

### 8.2.4 Data access

These tests will test the data access layer and the data layer. It will be necessary to have access to an SQL Server database to run the tests.

We will only describe the tests for the relational database-based data access implementation, but they should also be implemented and run for the XML-based data access implementation.

Again the bigraph  $A$  from section 8.2.2 are used.

#### T4.1 Persist a bigraph

Create a `BigraphDatabaseProvider` instance and call the `PersistBigraph` method with  $A$  as the argument.

#### T4.2 Load a bigraph

Call the `LoadBigraph` method on the previously instantiated `BigraphDatabaseProvider` instance.

Check that the bigraph is as expected. If this is the case both this and the previous test case are successful.

#### T4.3 Persist a composite bigraph

Create a `BigraphDatabaseProvider` instance and call the `PersistCompositeBigraph` method with the previously instantiated composite bigraph as the argument.

#### T4.4 Load a composite bigraph

Call the `LoadCompositeBigraph` method on the previously instantiated `BigraphDatabaseProvider` instance.

Check that the composite bigraph is as expected. If this is the case both this and the previous test case are successful.

## 8.3 Test results

The result of the tests can be seen in figure 8.1. Furthermore, the test case implementations and output can be found in the appendix section. The tests exposed one bug.

The exposed bug was found during composition of bigraphs. After the two bigraphs has been composed, the control references on the nodes that had been moved were still pointing to the old controls.

This is a small bug that can be fixed simply, so we conclude that the framework implementation functions according to the specifications.

As mentioned earlier, the tests performed in the context of this thesis would not be sufficient for a production system, but for a prototype they are quite adequate.

Test case	Result as expected?
T1.1	Yes
T1.2	Yes
T1.3	Yes
T1.4	Yes
T1.5	Yes
T1.6	Yes
T1.7	Yes
T2.1	No
T2.2	Yes
T2.3	Yes
T3.1	Yes
T3.1	Yes
T4.1	Yes
T4.2	Yes
T4.3	Yes
T4.4	Yes

Figure 8.1: The test results.

## 8.4 Summary

The tests described here do not cover all of the code and in a production system these tests would certainly be inadequate. However, as we consider the taxonomy framework a prototype, they are quite sufficient in this context. We can conclude that the implementation seems to work as it is designed to.

## Chapter 9

# Conclusion

In this chapter we will discuss the conclusions we have reached during the process of writing this thesis. We will discuss the major decisions we have taken, and how those decisions influenced the final outcome of the framework.

### 9.1 Conclusion

We started out with the goal of designing and implementing a prototype for a new version of the Taxonomy Tool, a tool built for working with taxonomies of generic objects.

Instead of just making a tool, however, we decided to make a framework for working with taxonomies. Subsequently, such a framework could be used for many different applications, one of which could be the tool mentioned above.

We came up with the idea of basing the framework on bigraphs, a mathematically founded model of tree structures with links between them. Besides being mathematically founded, bigraphs also had some unique features for manipulation of the structures - composition and reaction rules.

Based on real life use cases we defined a number of design goals and requirements. These would later help us design the framework because we had a very clear idea of what we wanted the framework to look like and be capable of.

Although we had decided from the beginning to base the framework on bigraphs, we decided it would be a good idea to survey what other data structures the framework could be based on. This could provide us with inspiration for features and also export/import formats.

The OWL data format could have provided the framework with an advanced type system, one of the defined requirements. However, it had nothing similar to the compositionality and dynamics of bigraphs.

We then started designing the bigraph-based taxonomy framework. We designed and implemented a multi-tiered framework for working with and persisting taxonomies. We were careful to design the data structures according to bigraph theory, and then provide necessary extensions on top of that. We will now evaluate what advantages and disadvantages employing bigraphs in the taxonomy framework gave us.

### 9.1.1 Bigraph evaluation

To satisfy the defined requirements we had to provide a few extensions to the theory, but this would have been necessary no matter what data structure we had chosen, unless we had decided to design a completely new data structure from scratch. Bigraphs did actually provide quite a few of the features required, such as tree structures and links with multiple participants.

As we were designing the framework it quickly became apparent that the most complex parts were composition and reaction rules, which is also what happens to be what sets bigraphs apart from other similar data structures.

In addition to being a complex operation in itself, composition is what necessitates regions, holes, and outer and inner faces. Implementing reaction rules is so complex that we decided not to attempt a full implementation of this.

Without this functionality bigraphs would simply be tree structures with links between them, which, incidentally, very well describes the basic requirements of the taxonomy framework. It can be said that it would have been easier to design a framework fulfilling the requirements if it did not include the compositionality and reaction rules functionality.

We should therefore ask ourselves if providing functionality such as composition and reaction rules is worth the complexity it brings. The current Taxonomy Tool does not have similar functionality, but we have suggested use cases where especially compositionality could be very useful. Particularly because this functionality allows for sharing of structures do we consider it to be useful in many scenarios.

Examining how the complex functionality of compositionality and reaction rules could be employed in a variety of settings is beyond the scope of this thesis. However, we will conclude that there are indications that such functionality would indeed be useful in some scenarios. The experience we have gained from our implementation, however, suggests that the process of creating a practical tool based on bigraph theory is indeed a complex task.

The complexity arises when one attempts to apply the formal mathematical models to a real life environment. Obviously, the mathematical models work well in theory, but an actual implementation must also take into consideration such notions as persistence of data, available resources and limitations in programming languages.

The question is whether the final application actually benefits from being based on a formal mathematical model? Would it have been more advantageous to implement similar functionality in a different way that did not necessarily adhere to the formal model?

In general, we can certainly see an advantage in adhering to the formal model in some cases. An example is concurrency models in distributed systems, where strict adherence to the formal model is necessary for the system to work correctly. Another example is database system implementations which are based on formalized mathematical models.

In general, a formal model also offers the advantage of a model description independent from the implementation. If such a model does not exist, the implementation in many cases end up also being the documentation. A formal model can function as concise documentation, so that users can understand precisely how an application works internally. This can often help clarify whether or not an application is suitable for a specific task.

However, in the case of bigraphs it is unclear whether adhering to the formal models really provide us with any advantages. We believe that similar functionality could be implemented without doing this, and therefore conclude that adhering to the formal model has not been an advantage for us.

There are several alternatives to doing things the way we chose to do it.

It is certainly possible to design an application featuring the same functionality as bigraphs, but which is not based on the formal model of bigraphs. Had we chosen to do this we could probably have avoided details such as composite bigraphs, which was implemented because the formal model was not compatible with the specified requirements.

The formal definitions are not set in stone, and can be altered if this is found to be practical. Already there exist several different sets of definitions, the one we used simply being the one referenced most often. Therefore, it might be possible to create a set of definitions that are better suited for implementation.

It might also be that other programming paradigms are better suited for implementing formal mathematical models than object oriented languages, of which *C#*, which we employed, is a representative. Functional programming languages, which are based on mathematical functions, would be an obvious contender here.

Testing these alternatives, however, is outside the scope of this thesis.

In conclusion we will say that our implementation does fulfill the specified requirements while adhering to the formal model of bigraphs to a large degree.

We believe that the combination of a generic taxonomy tool and bigraphs results in a very interesting and useful tool, that could be employed in a variety of settings. Workflow applications is an example of this.

However, strictly adhering to the formal model of bigraphs in our implementation has dramatically increased the complexity of the implementation, and we believe a better solution could have been attained by pursuing one or more of the alternatives described above.

## 9.2 Perspectives

We will now discuss a number of perspectives that we have formulated on the background of writing this thesis. These perspectives can be divided into perspectives relating to the taxonomy framework and perspectives relating to bigraphs.

### 9.2.1 Taxonomy framework

In this thesis we have defined a number of requirements that should be satisfied by a taxonomy framework implementation whether or not that implementation is based on bigraphs. Although we did not cover all requirements in the analysis chapters of this thesis, they are all very valid requirements in the context of a full implementation.

Especially the type system requirement which dictates that nodes as well as links should be typed, and it should be possible to define restrictions as to what types of nodes can be vertically related and what types of nodes can be connected to what types of links is a relevant requirement. As we have seen in our brief description of OWL this can be achieved in different ways.

Bigraphs add an extra dimension to the discussion of type systems, in that bigraphs can be characterized as typed tree structures with links. It would be simple to design a tree structure where it is possible to append a subtree to a node. But bigraphs take this one step further and allow us to define interfaces for tree structures. Instead of appending two nodes, which is conceptually a relatively simple operation, we can compose trees, and in doing so we can also merge links between the trees. The interfaces define what bigraphs can be composed, and therefore they can be said to define the type of a bigraph.

We predict that the definition of a type system will play a large role in the final design of a taxonomy framework, and it is clear that bigraph theory can contribute to this.

### 9.2.2 Bigraphs

Two of the main objectives of this thesis has been to attempt an implementation of bigraphs and to employ that implementation in an application. The purpose of this has been to evaluate how suited bigraphs are for this.

The overall conclusion is positive. We have argued that the basic structure of bigraphs makes the theory applicable for a number of scenarios, and specifically the taxonomy framework seems to be an obvious candidate as an application that could be based on bigraphs.

However, we have encountered some shortcomings in the theory which we think it would be reasonable to attempt to address in future updates of the theory.

We encountered these shortcomings specifically because we have attempted a practical implementation of the theory, which forced us to think about bigraphs in a different manner than is usually done when working with the theory. If bigraphs is to be employed in practical solutions in the future, e.g. in the context of ubiquitous computing, we believe it will be necessary to address such shortcomings.

As discussed in chapter 5 we found the lack of sharing of bigraph structures to be a shortcoming in the context of the taxonomy framework, and we also predict that sharing of structures will be a requirement in many of the other applications that could potentially be based on bigraphs, e.g. the workflow patterns use case.

The composite bigraphs that we have defined in this thesis is one suggestion as to how sharing can be provided.

Another way of providing sharing could be to designate some elements as copyable and others as uncopyable, so that if a bigraph is copyable it can be shared.

A third solution might be inspired by the type system of OWL, where it is possible to create classes and then instantiate these. This would enable users to define reusable structures as classes and then instantiate these when needed.

Another shortcoming is the lack of a generic way of transforming bigraphs. Composition is a powerful feature, but it is not practical for simple transformations, e.g. attaching a node to another node. Also, some transformations, e.g. removing a node, cannot be achieved by composition. Although all possible transformations can be achieved by reaction rules, reaction rules are not really suited as a generic way of transforming bigraphs either because it is necessary

to define an instance of a reaction rule for each transformation. This implies the need for a more generic way of transforming bigraphs.

Another shortcoming is the lack of a way of querying bigraphs. In our implementation we have tried to implement this by defining reaction rules with return values, but this is hardly a good solution. If bigraphs are to function as data containers, e.g. in the context of ubiquitous computing, it should be possible to query bigraphs. A good example of a data container format with powerful querying tools is XML which has both XPath and XQuery defined as query languages. XML is a simple technology, but one of the reasons that it is useful as a data container is because it has these query languages.

A bigraph query language could be designed to provide both transformation and querying capabilities (XQuery is designed this way), thus addressing both shortcomings mentioned above. This would enable searching through bigraphs and performing simple transformations.

Finally we will comment briefly on the type system in bigraphs. The taxonomy framework required a more complex type system than what bigraphs had to offer. This implies that there might be a general need for a more complex type system in bigraphs, something which Milner also has suggested as mentioned earlier. Such a type system could be inspired by what we have implemented in this thesis as that implementation does not stray too far from the established theory.

# Bibliography

- [1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.
- [2] E. R. Harold and W. S. Means. *XML in a Nutshell, Second Edition*. O'Reilly & Associates, Inc., 2002.
- [3] T. Hildebrandt and J. W. Winther. Bigraphs and (reactive) xml - an xml-centric model of computation. Technical report, IT University of Copenhagen, 2004.
- [4] O. H. Jensen and R. Milner. Bigraphs and mobile processes (revised). Technical report, University of Cambridge, 2004.
- [5] R. Milner. Axioms for bigraphical structure. Technical report, University of Cambridge, 2004.
- [6] R. Milner. A scientific horizon for computing. <http://www.cl.cam.ac.uk/users/rm135/beijing2005.pdf>, 2005.
- [7] W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow patterns. <http://is.tm.tue.nl/research/patterns/download/wfs-pat-2002.pdf>, 2005.
- [8] W3C. Owl web ontology language guide. <http://www.w3.org/TR/owl-guide/>, Feb. 2004.
- [9] W3C. Owl web ontology language overview. <http://www.w3.org/TR/owl-features/>, Feb. 2004.
- [10] W3C. Owl web ontology language use cases and requirements. <http://www.w3.org/TR/webont-req/>, Feb. 2004.
- [11] W3C. Rdf primer. <http://www.w3.org/TR/rdf-primer/>, Feb. 2004.
- [12] W3C. Document object model (dom). <http://www.w3.org/DOM/>, Jan. 2005.
- [13] J. W. Winther. Reactivexml. Master's thesis, IT University of Copenhagen, 2005.