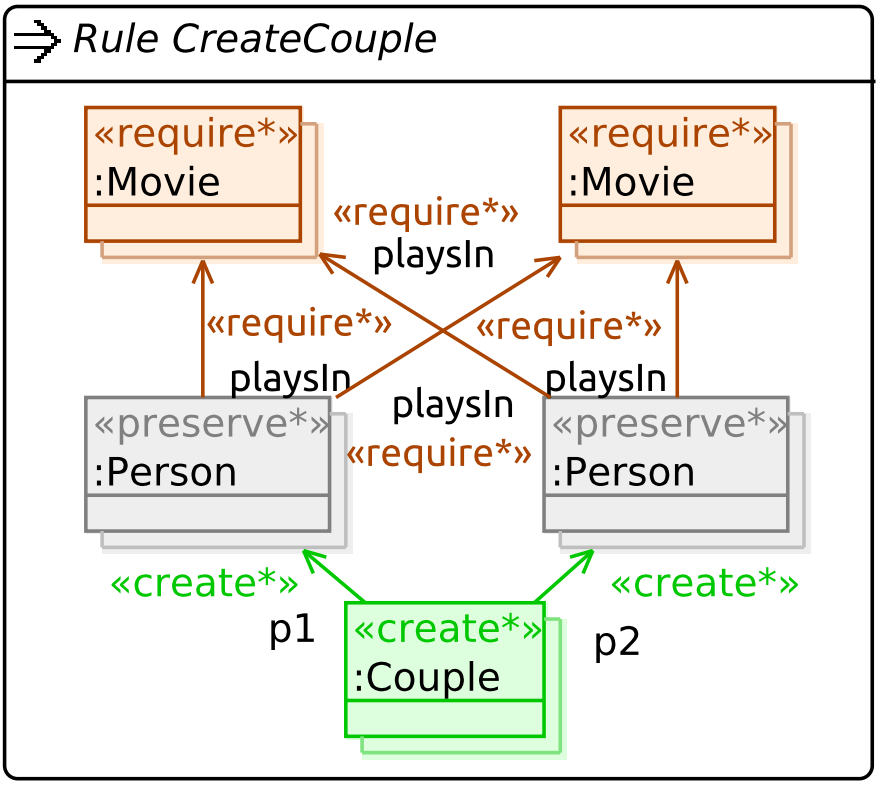
CHALMERS



Improving the performance of graph transformation execution in the BSP model

By improving a code generator for EMF Henshin

Master of Science Thesis in Software Engineering

FREDRIK EINARSSON

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Gothenburg, Sweden, June 2015

Master's thesis 2015:NN

Improving the performance of graph transformation execution in the BSP model

By improving a code generator for EMF Henshin

FREDRIK EINARSSON

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Division of Software Engineering
Chalmers University of Technology
Gothenburg, Sweden 2015

Improving the performance of graph transformation execution in the BSP model
By Improving a code generator for EMF Henshin
FREDRIK EINARSSON

Supervisor: Matthias Tichy, Department of Computer Science and Engineering
Examiner: Miroslaw Staron, Department of Computer Science and Engineering

Master's Thesis 2015:NN
Department of Computer Science and Engineering
Division of Software Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: The picture shows the graphical representation of the Henshin transformation rule *CreateCouple*. The rule, and thereby the picture, is adopted from the original created by Krause et al [1].

Typeset in LaTeX
Printed by [Name of printing company]
Gothenburg, Sweden 2015

Improving the performance of graph transformation execution in the BSP model
By improving a code generator for EMF Henshin
FREDRIK EINARSSON
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

Model-Driven Engineering is widely used within Software Engineering to increase the level of abstraction and thereby increase the efficiency and maintainability of software projects. Combining the notion of Graph Transformations with the Bulk Synchronous Parallel (BSP); big-data problems can be solved using a model-driven approach. However, such solutions are still in their early phases and do not provide the required performance. This presents the performance issues with a current approach for executing Henshin transformation models on a Hadoop cluster utilizing the BSP implementation Apache Giraph. A solution, in the form of an improved code generator, is developed in order to address the problems. The improved approach is evaluated using a purposefully designed set of synthetic graphs, transformations rules and a real data example. The real data example is the IMDB movie database and the evaluation is performed on a compute cluster. The results indicate that the new solution performs better and uses less memory for the synthetic graphs and rules with particular characteristics while it in this particular real data example shows no significant difference.

Keywords: incremental graph pattern matching, graph transformations, bulk synchronous parallel, cluster benchmark

Acknowledgements

I would like to thank my supervisor Matthias Tichy for continues input and high-class supervision throughout the thesis work.

Fredrik Einarsson, Gothenburg, June 2015

Contents

List of Figures

List of Tables

1

Introduction

1.1 Background

Model-Driven Engineering (MDE) is a term used within the area of Software Engineering. Kent [2] devised the term back in 2002 and since then numerous publications has been published on the subject. The term presses the usage of abstract models as a central artifact within the development of software. Models evolve together with the software and is used both for the understanding and the detailed design; enabling an increased level of abstraction in order to increase the efficiency and maintainability of software project. Model-Driven engineering also, among other usages, enables increased automation via model transformation and code generation.

Big data is a broad and widely used term for expressing the exponential growth of data. The term was first coined by Doug Laney [3] and has since then exploded in usage. An approach for tackling these large datasets is graph-based storage, analyses and manipulations. The relevance and importance of a graph-based approach has increased with the growth of services such as social networks (for example Facebook) and context-aware search, as well as continuously being valuable in applications such as product lifecycle management (PLM) and supply chain management (SCM) [1]. Such services store large amount of data in the form of graphs and as graph sizes grow, the need for searching and manipulating these with high performance and low resource utilization increases. To achieve high performance, parallelization of the execution is a viable approach, but also puts extra concerns on the programmer.

Continuing the tradition of Model-Driven Engineering by putting the model in center and using a graph-based approach for solving the problem of increased data volumes, the theory of algebraic graph transformations [4] can be used. Graph transformations provide a highly abstract modelling concept in order to address the issue of performing highly efficient, parallelized execution of manipulations on graphs, while still preserving high expressiveness [1]. Also enabling to address the problem of increased complexity and maintainability with the increased size of software and data.

A technology which can be used for solving such problems is the Eclipse Modelling Framework (EMF) [5]. EMF is a modelling framework and code generation toolbox

for building Java applications based on structured data models [6]. EMF offers a plethora of tools related to modelling. One of those, Henshin [7], provides a toolset and a model transformation language with formal graph transformation semantics [4] which allows formal reasoning. Henshin further offers amalgamation transformation units which are units especially suitable for parallel application of transformation rules by use of one of its operators [6].

Some existing solutions utilize EMF for addressing the problem of large graphs and the need for parallel execution with high performance, e.g. [1], [8]. In [1], Krause et al. present their solution, which builds on Henshin. Their solution uses Henshin to create high-level graph transformations with the expressiveness of algebraic graph transformations and a code generator for generating runnable code from the transformations to execute on a cluster. The solution addresses the problem of parallel execution of transformations with high performance on graphs with millions of nodes. It also scales, horizontal and vertical. However, with growing data sizes, it is required to add additional computation nodes. It is then also crucial to take extra care in the efficiency and resource utilization of the algorithm which is currently a limiting factor [1].

To address the demand of high performance on large graphs the code generator generates code adapted for the Bulk Synchronous Parallel (BSP) abstract machine [1]. BSP provides a bridging model for designing parallel algorithms which can be executed on distributed computer systems [9]. By mapping graph transformations to this model, graphs with millions of nodes can be iteratively searched and transformed [1].

A technology which is inspired by BSP is the iterative graph processing system Apache Giraph [10]. Apache Giraph is used by Krause et al. [1] as a target platform for their code generation and is built for high scalability, used at Facebook and adds several additional features [10]. The open source software Apache Giraph builds upon the MapReduce programming model [11] implementation Apache Hadoop [12]. Another similar BSP inspired graph processing framework is Pregel which is developed by Google [13].

1.2 Statement of the Problem

As stated in Section 1.1, Krause et al. [1] have seen some limitations to their approach. Above a certain number of added computation nodes the execution time does not decrease with the same rate when adding new nodes to the cluster (scale horizontally). This can be seen in Figure 1.1 which shows their results when executing a benchmark on the generated code on a cluster. In the benchmark, the dataset is constant and the number of nodes is the independent variable while the dependent variables are the execution time and the total used node time. Looking at the figure, when the number of nodes reach approximately 14 nodes the benefit in shorter execution time when adding more nodes starts decreasing while the total used node time keeps increasing.

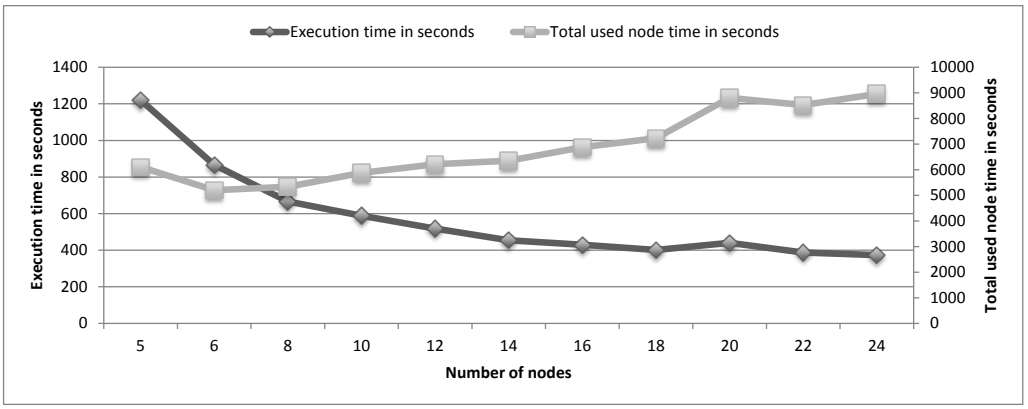


Figure 1.1: The figure illustrates the execution time and the total used node time of a graph transformation rule on a millions of vertices large graph in the evaluation performed by Krause et al [1].

As the data sizes grow further the need for fast execution time and low resource utilization can be expected to increase. To achieve this the solution must have better scalability properties. Therefore, there exists a need to further improve the scalability of the solution provided by Krause et al [1]. This while still preserving the expressiveness of a declarative high-level modelling language, which abstracts away the basic graph operations.

1.3 Purpose of the Study

The previously described performance problems of Krause et al's [1] implementation are hypothesized to be due to the requirements of communication bandwidth and memory. For further details see Chapter 5. By investigating, implementing and benchmarking a different approach for limiting the communication needs and the memory consumption; a higher rate of performance increase when increasing the number of computation nodes (higher horizontal scalability) could possibly be gained. The purpose of this study is thereby to investigate and document the suitability of an alternative approach. By doing so, the overall understanding of what affects the performance of graph transformations in the Bulk Synchronous Parallel model can be deepened, possibly resulting in the improvement of current approaches.

The aim of this thesis is not only to find the best overall approach, if such exists, but also to find in which situation the solution has the highest respective lowest benefits. To achieve this a set of synthetic data in the form of graphs and transformation rules, which can be used as a benchmarking suite, will be created. The purpose is therefore also to create, document and present this benchmarking suite for future usage as a reference set for future improvements.

The study will attempt to further develop the available work in this area and therefore the researchers involved in the area will benefit from this thesis bringing the

state of the art forward. If the results are beneficial they will also be integrated into the next Henshin [7] version which enables usages outside the academia. It could then also be used for other current distributed graph applications. An example of an application is the execution of transformations on the huge graphs of the gathered user data in social networks.

1.4 Overview of the Results

This section aims to provide a short overview of the results of this thesis. They consists of four main parts. A qualitative analysis of the problems of a current solution for executing graph transformations in the Bulk Synchronous Parallel model is conducted. A benchmark is developed in order to evaluate a new solution. The solution is designed, implemented and presented and aims to solve some of the found problems. Finally, a comprehensive evaluation of the new solution is conducted were it is compared to the existing approach.

The qualitative analysis resulted in a number of problems were the problems with utilizing Bulk Synchronous Parallel (BSP) model and the procedure of mapping graph transformations to the BSP model are the two most important. BSP is highly parallelized however rigid in its conceptual structure. The same type of actions are performed for all vertices throughout the graph in the same time frame. This leads to bottlenecks in a computation.

The mapping from graph transformations to the BSP model inherits the problems with the BSP model without taking actions to reduce them. The algorithm consists of a series of microsteps which iteratively builds up and passes around a set of partial matches. These microsteps are executed for each vertex of the graph and requires thereby the same kind of resources. The same yields for the partial matches which are all of the same type and size in a particular superstep.

The designed and developed improved version of the code generator and mapping of graph transformations to the BSP model takes into the found problems and develops a version which attempts to spread out the requirements of resources throughout the computations supersteps. It does so by decoupling the microstep from the computations superstep and thereby spread out the microstep executed and type and size of partial matches.

A benchmarking procedure is developed in order to evaluate the solution against the existing one. It consist of metrics, source graphs and transformation rules. The metrics aims to meassure how the solution performs with a focus on execution time and memory consumption. The source graphs consists of a real data example and synthetic graphs. The real data example aims to see how it works in a particular real data example and the synthetic graphs aims to stress test the solution. The transformation rules also consisted of a real data example and a set of purposefully designed rules. These had the same purpose as the source graphs.

Lastly, an evaluation was performed on the new implementation utilizing the devel-

oped benchmark on a compute cluster. The real data example was used to evaluate the execution time using a number of different amounts of cluster nodes. There was no significant difference in terms of execution speed when the two implementations were performed. However utilizing the synthetic graphs and transformation rules the new implementation performed significantly better in terms of both execution time and memory usage. It was especially suitable for the case of source graphs with a high out-degree and transformation rules with many microsteps. What these have in-common is a high amount of partial matches generated which requires large amount of memory and bandwidth. The new solution thereby succeeds in even out the resource requirements and improves the performance in some cases.

1.5 Disposition of the Report

Chapter 1 has described the overall background, the problem as well as the purpose of the report and its significance. The following parts will in order discuss the following. Chapter 2 describes vital theoretical background as well as technology used in the thesis. Chapter 3 goes into detail about the research method, the research questions and the limitation and delimitation of the thesis. Chapter 4 describes related work used both as a foundation for this thesis as well as other related work which has similarities to this work. In Chapter 5 the previous solution is analysed to try to find the limiting factor. In Chapter 6 the benchmarking suite created is derived and described. Chapter 7 discusses the new solution while Chapter 8 evaluates the new solution using the benchmark. In Chapter 9 potential threats to validity are presented. Finally Chapter 10 provides the final conclusion and provides suggestions for future work.

2

Foundations

This chapter describes the theoretical concept as well as the used technology. The description is in general brief, however some parts of extra importance for the understanding of the developed solutions are presented more in-depth. The chapter includes the following:

In Section 2.1 the Eclipse Modeling Framework is presented. In Section 2.2 the theory of graph transformations is presented. Based on the beforehand description of graph transformations and the Eclipse Modelling Framework, Section 2.3 describes the specific semantics of Henshin. In Section 2.4 the Bulk Synchronous Parallel model is described to later in Section 2.5 present the Bulk Synchronous Parallel implementation Apache Giraph and its special characteristics and differences compared to the standard Bulk Synchronous parallel model. Lastly, Section 2.6 describes how the for Henshin created code generator maps the Henshin models to the Giraph computational model.

2.1 Eclipse Modeling Framework

Eclipse Modeling Framework (EMF) [5] is an extension upon the well-known Eclipse Java Integrated Development Environment [14]. EMF includes tools for creating, using and maintaining Java applications built on structured data models. Features central for this thesis includes the meta-modeling language Ecore and code generation facilities.

Ecore builds on Meta-Object Facility (MOF) and enables the creation of meta models using a graphical syntax similar to UML class diagrams. It has syntax for e.g. specifying entities, attributes, associations, types, inheritance and multiplicity. An example is the movies meta-model (Figure 2.1), which is used throughout this thesis. The meta-model is used for defining model instances which stores information about movies, persons acting in movies, and persons commonly playing in the same movies (*Couples*). The meta-model has entities (e.g. *Person*), attributes (e.g. *rank*), associations (e.g. *occursIn*), uses inheritance (e.g. *Actress* and *Actor* are *Persons*), multiplicity (e.g. a *Couple* occurs in zero or one *Movies*) and entities and attributes are typed.

EMF is easy to extend, and therefore a plethora of plugins have emerged. Those

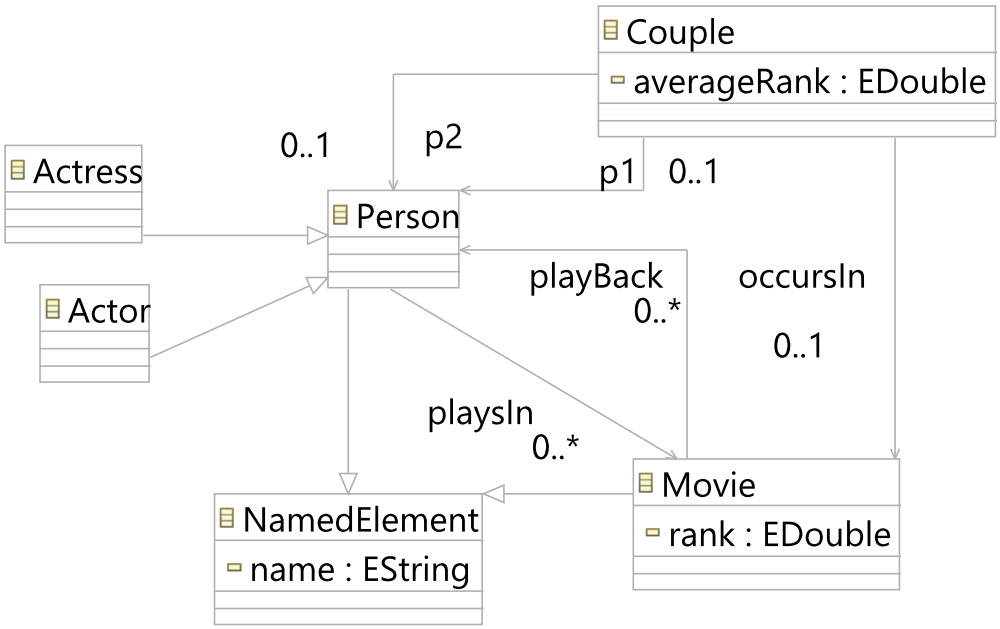


Figure 2.1: Movies meta-model. Derived from the work of Krause et al [1].

include Java Emitter Templates (JET) [15] model to text transformation language, which is used to create the code generator (Section 2.6), and Henshin, which is described in Section 2.3. JET enables mixing of text and Java code for iterative and conditional generation of content of text files. The Java code can for example hold the Ecore models and the graph transformation rules and generate executable Java code depending on their structures.

2.2 Graph Transformations

This section provides a brief introduction to the concept of graph transformations. Especially it explains the part needed to understand the rest of the thesis. For a complete overview see e.g. Ehrig et al. *"Fundamentals of Algebraic Graph Transformation"* [4].

Graph transformations provide abilities to define declarative and operational changes of data, stored in the form of graphs, in a high-level of abstraction [1]. *Declarative* relates to specifying what needs to be achieved and *operational* how that should be achieved. Graph transformations can therefore be used to increase the programmer's productivity by abstracting away basic graph operation (such as traversals) and thereby provide high expressiveness.

Graph transformations can be applied on both typed and untyped target graphs, but since Henshin (see Section 2.3) uses types for both vertices and edges, only typed graphs are considered here. The same applies to directed and undirected graphs where Henshin works with the former. However, these two decisions are not limitations since they can be seen as more expressive and carrying more data than

the opposite choice. The choice can also be used for creating the opposite choice, e.g. edges in both direction gives bidirectional edges.

A graph transformation rule, p, consists mainly of two parts and can be formally defined as $p = (L, R)$, where L is called the left-hand side (LHS) and R is called the right-hand side (RHS). To *apply* the rule means to find a match for the LHS in the source graph and replace it by the RHS [4]. In order to do so vertices of the LHS are mapped to vertices of the RHS.

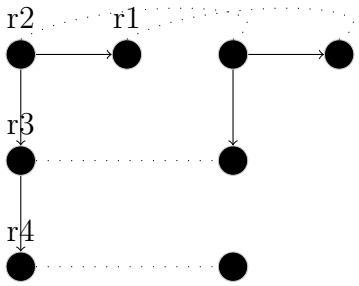
According to Ehrig et al. [4] is the main practical problem how to perform the replacement. For example, if the LHS carries more data than specified in the rule or if a vertex in the rule is of a supertype of a vertex in the source graph it cannot simply be replaced without causing unexpected behaviour. Recreating object can also be inefficient for large graphs. However, in Henshin (see Section 2.3) and the written code generator (see Section 2.6) the match in the source graph is instead transformed to fulfill the requirements of the RHS.

Figure 2.2 provides an example of a simple graph transformation rule, where all vertices and edges are of the same types, and the execution of it. Figure 2.2a illustrates the rule with its LHS, located to the left, and its RHS, located to the right, with mappings, represented with dotted lines, in-between. The LHS consists of four vertices and three edges and the RHS of four vertices and two edges. All vertices and edges of the RHS are mapped to vertices of the LHS. The rule thereby represents the removal of one of the edges. Figure 2.2b represents the source graph on which the rule is applied. The execution of the rule yields two matches. The vertices *r1*, *r2*, *r3* and *r4* in the LHS of the rule matches the vertices *d*, *a*, *b* and *c* respective *g*, *d*, *e* and *f* in the source graph. Two matches yields two rule application and thereby two edges are removed resulting in the target graph of Figure 2.2c.

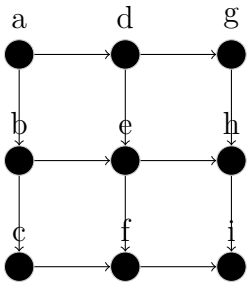
The rule of Figure 2.2 is simple; containing only the most basic features, and in-short only describes the look of the LHS and RHS and how they map. In a real world problem there is a need to define more complex structures. Therefore rules are made even more expressive by adding types, constraints for the structure of the LHS and RHS and constraints for attributes. Therefore the LHS and RHS can be further broken down using those features.

The LHS describes under which conditions the rule should be applied. How a part of the graph should and should not look (constraints) in order to be considered a match. Constraints consists of application conditions which specify the required characteristics. The requirement can be in the form of specific types and numbers of edges connected, particular adjacent types and number vertices, specific values of attributes or particular types of the vertices and/or edges contained. The application conditions can be either positive or negative application conditions (PAC or NAC). The PAC specifies what should be there and the NAC what should not. Both have to be fulfilled for the sub-graph to be considered a match. It is here also possible to specify if a vertex or edge should be a part of the actual match or if it is just a condition.

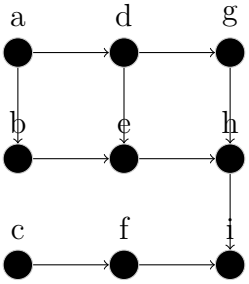
On the other hand, the RHS describes how a match for the LHS should look after



(a) The rule $p = (L, R)$. The LHS (to the left) and RHS (to the right) with mappings in-between.



(b) The source graph.



(c) The target graph

Figure 2.2: The figure illustrates a graph transformation rule $p = (L, R)$. Figure (a) represents the transformation rule executed on all matches in the source graph (b) resulting in the target graph (c).

the transformation rule is applied and thereby indirectly what should be done with the match when it is found. It can describe the creation, deletion or alteration of edges, vertices or attributes. Thereby graph transformation execution consists in short of two steps. First a match is found and then the rule is applied on the match to satisfy the RHS. The key here is that the programmer does not specify how this should be done but only what should be done (declarative vs operational). The engine executing the transformation or the code generator (see Section 2.6) generating the running code needs to take care of this.

A match can be *injective*. In an injective match a vertex in the source graph can only map to one vertex in the LHS. This irrespective of how many vertices in the LHS the vertex fulfills the conditions for. If a match is not injective, the vertices in the source graph can be mapped to several vertices of the LHS. Henshin (see Section 2.3) defaults to injective matching and therefore the rest of the discussion assumes injective matching.

2.3 Henshin

The Henshin project [7] is a model to model transformation language for the Eclipse Modelling Framework (see Section 2.1). Henshin handles model instances of meta-models specified using Ecore (see for example Figure 2.1). It supports both endogenous (source and target meta-models are equal) and exogenous (source and target meta-models differ) transformations. Since the current implementation of the Henshin code generator for Giraph (see Section 2.6) only works with endogenous transformations, exogenous transformations are not considered further.

The Henshin project has additional features apart from the core transformation language. Such features are a graphical representation of the rules (see Figure 2.3) and a graphical rule editor; which makes it easy to specify, discuss and maintain transformations, a tree based editor; enabling more detailed specification, the local transformation engine; enabling local execution of transformations on a single machine and the code generator for Apache Giraph [10]; enabling execution of transformations in a distributed environment (see Section 2.6).

The model transformation language and the semantics of Henshin is based on the theory of graph transformations (see Section 2.2). This enables the possibility of formal reasoning [1]. However, although it is based on the theoretical concept it adds more features and also has some extra complications. Regarding the extra complications, Krause [16] has for example noted that model transformation cannot be seen as the same as graph transformations. The reason is the usage of meta-models which can cause unexpected behaviour in terms of side-effects. For example can an edge be removed without this being explicitly specified in the rule. The reason is that the meta-model only allows a certain number of edges and in order to create a new edge an old edge must be removed. However, the code generator (see Section 2.6) has non of these side-effect and can therefore be considered pure graph transformations.

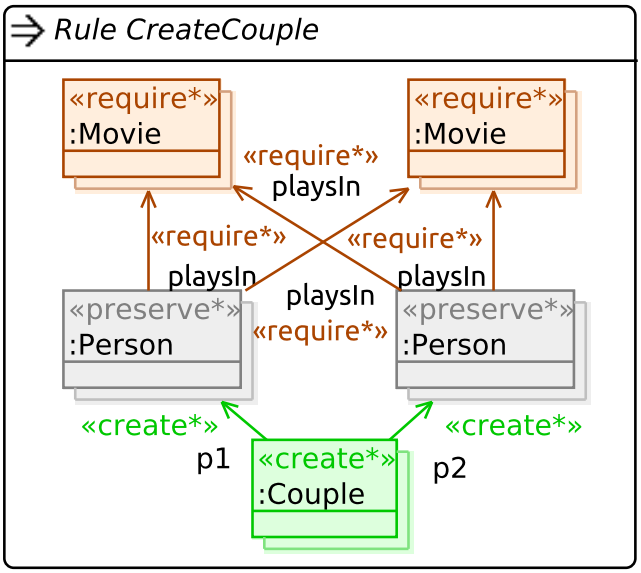


Figure 2.3: The graphical representation of the Henshin transformation rule *CreateCouple*. The rule is adopted from the work of Krause et al [1].

The language supports both the declarative rules and the operational units as explained in Section 2.2. Declarative rules shows what needs to be achieved without specifying how and in what order. Operational rules gives the ability to specify order of operations and rules.

Declarative transformation rules are in Henshin specified using the keyword *Rule*. Figure 2.3 shows the graphical representation for a rule called *CreateCouple*. Apart from specifying the structure of vertices and edges, a rule can also be extended with stereotypes and provide conditions for the attribute values [6]. These stereotypes can be seen in Figure 2.3 enclosed by *«* and *»*. Stereotypes make up the PAC (positive application condition) and NAC (negative application condition), as explained in Section 2.2, as well as the LHS (left-hand side) graph and RHS (right-hand side) graph of the transformation rule. Stereotypes, which can be placed on both edges and vertices, can be used to specify the preservation, creation or deletion of nodes (see e.g. *«create*»* and *«preserve*»* in Figure 2.3). When they are used for specifying the application conditions they can be either *require* or *forbid*. *Require* specifies that a certain vertex or edge should exist however it is not a part of the final match. *Forbid* tells the execution engine that a match cannot have a certain edge or vertex connected to it. These stereotypes can also be combined using boolean operators to create more complex application conditions.

In Figure 2.3 stereotypes (*create* and *require*) can be seen for both edges and vertices. The two vertices of type *Person*, annotated with *«preserve*»*, indicates that they should be a part of the LHS and also a part of the final match while the two vertices annotated with *«require*»* annotates that they should be a part of LHS however not a part of the final match. The difference is that a final match should be unique independent on how many times the rest of the LHS is matched. For example, two

vertices of type *Person*, satisfying the requirements of the LHS, should only result in one rule application irrespective of the number of times they satisfy the rest of the requirement. The RHS, on the other hand, includes one extra vertex and two extra edges which is annotated with *«create*»*.

For enabling maximum parallel execution a rule can also consist of on or many *multirules*. A multirule is by default executed on every found match instead of the default behaviour which is on the first match found. The rule in Figure 2.3 contains one single multirule with all the vertices and edges included. This is shown via the asterisk in the stereotypes (e.g. *«create*»*). More advanced structures can be created using the multirules, however there is currently no graphical syntax for this and the Giraph code generator (see Section 2.6) has no support for more than one multirule. In that case the multirule needs to contain all vertices and edges.

Henshin's operational transformation units are on the other hand used for operationally defining control-flows [6]. An Operational unit contains one or many other rules and units. An operational unit can be either a sequential unit; for defining order of application, an iterative unit; for defining fixed number of iterations, a looped unit; for applying the rule until it is no longer applicable, an unit for non-deterministic execution, an unit for prioritized execution or an unit for defining conditional execution. Figure 2.4 shows the graphical syntax for an operational transformation unit of type sequential unit, called *AddCouples*, which contains two other rules. The *CreateCouple* rule (see Figure 2.3) and another rule called *CreateOccurrence* is contained in the sequential unit and should be executed in order.

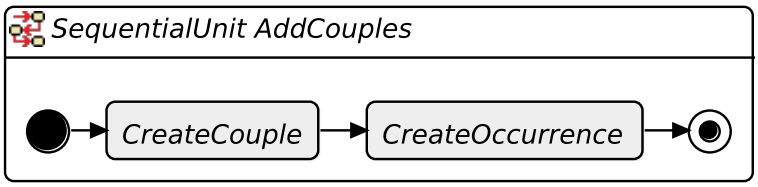


Figure 2.4: A sequential transformation unit called *AddCouples*. The unit is adopted from the work of Krause et al [1].

2.4 Bulk Synchronous Parallel

To understand the solution and the possible reasons for the performance issues of the current distributed implementation of Henshin it is important to understand the underlying computation model. Because the code generator (see Section 2.6) created by Krause et el [1], which this thesis aims to improve, outputs code to be executed using Apache Giraph, which is inspired by the Bulk Synchronous Parallel (BSP) computation model, the BSP model needs to be understood.

BSP was introduced by Valiant [9] as an aid for developing parallel algorithms. It is currently also an approach for efficiently processing large-scale graph data as can

be seen in for example the Giraph project (see Section 2.5). The approach is used for designing highly parallelized algorithms.

When designing the algorithms a specific scheme must be followed. The scheme consists of a series of supersteps where computations are performed locally with only local knowledge at each vertex in the graph. The scheme is illustrated by Figure 2.5. In the picture time passes downwards and each vertical silo represents a vertex. At the first superstep all the vertices are active. Each superstep consists of a computation phase and a communication phase which are organized in the following four steps:

1. Master Computation - The master node initializes the superstep by performing a single computation. The step can be used for bookkeeping and coordination of the computations performed in the vertices.

2. Vertex Computation - The active vertices perform local computations in parallel. Only local information is available during this step. During this step the vertex can modify the state of itself or outgoing edges as well as mutate the topology. Incoming messages from the previous step are available during the computation.

3. Communication - During vertex computations the vertex can send messages to other vertices (adjacent or by ID). The messages are received during the next superstep. Vertices can also be requested to be inactive for the next step.

4. Barrier synchronization - The local computations and the communication must be finished for every vertex before the next superstep can start. This is achieved via barrier synchronization. If it exists active vertices after this step, the execution is restarted with Step 1, otherwise it is concluded.

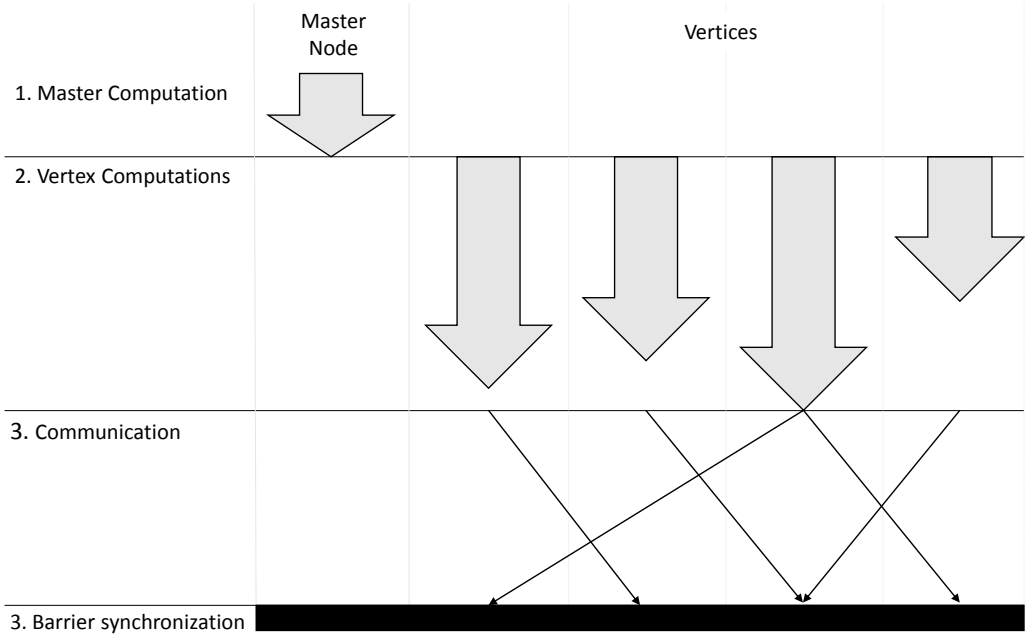


Figure 2.5: The figure provides an illustration of the BSP model.

2.5 Apache Giraph

Apache Giraph [10] is an open-source software for iterative batch processing of graphs. It is built to be highly scalable and follows in general the Bulk Synchronous Parallel computational model presented in Section 2.4. The Giraph project is highly modular, utilizing a large set of separately maintained projects such as the Apache project Zookeeper for orchestrating the distributed environment, log4j for maintainable logging and most importantly Hadoop as the distributed computation platform [17].

Giraph runs as an only-mapping Hadoop MapReduce job. The input graph is partitioned in a number of partitions. These partitions are distributed to a number of workers. One worker runs as a Hadoop mapper and can therefore be seen as a separate compute node. A physical node can contain one or many workers. A worker performs independent calculations apart from the message passing in-between each superstep and outputs a separate output partition in the end of the job. The partitioning is automatically handled and is not controlled by the BSP algorithm implementer. However the number of workers, partitions, compute threads per worker and amount of memory can be adjusted. This is however difficult to optimize and therefore set as a constant in the evaluation performed in this thesis.

In general a Giraph computation follows the four steps of the BSP model (see Section 2.4), however it adds some extra functionality which are useful in the development of the code generator (see Section 2.6) as well as in the evaluation (see Chapter 8) of it. In order to later being able to describe and discuss the code generator, the core functions and structure of a Giraph computation is discussed here. In order to do so the *AddCouples* unit in Figure 2.4 is used as an example.

A basic algorithm is in Giraph created by extending a *Computation* class, in this case the *BasicComputation* class. Listing 2.1 provides an example from the class *AddCouples*. Also note that the computation is supplied with four more type parameters. These are in order: the type of the id of the vertex, the datatype of the vertex data, datatype of the edge data and the message data type. In this particular example they are of type (*VertexID*), *ByteWritable*, *ByteWritable* and *Match*. How these are used in the specific case is discussed in Section 2.6.

Listing 2.1: The class header of a Giraph computation generated from *AddCouples*.

```
public class AddCouples extends BasicComputation<VertexId, ByteWritable,
    ByteWritable, Match> {
```

When extending a computation a set of methods becomes available and can be further extended. Step 2 (Vertex Computation) of the BSP model (see Figure 2.5) is executed for every vertex of the graph. This is achieved by extending the *compute* method (see Listing 2.2) and the underlying software ensures that the snippet is executed for every vertex at every superstep. In order to perform the calculation the method has two input parameters: The current vertex and a list of incoming messages. The vertex has the type parameters id type, vertex data type, edge data type and incoming messages type. These types are equal to the ones in the class

header (see Listing 2.1). In the *compute* method messages can be registered for sending. The message passing is the third step of the BSP algorithm and is here performed seamlessly in the background after a message is registered.

Listing 2.2: The *compute* method header of *AddCouples*.

```
@Override
public void compute(Vertex<VertexId, ByteWritable, ByteWritable> vertex, Iterable
    <Match> matches) throws IOException {
```

Giraph also adds methods which are guaranteed to run before and after each computation. Connected to the BSP model these can be seen as step 1.5 and 2.5. These methods are the *preSuperstep* method and *postSuperstep* method and is executed one time per partition. Listing 2.3 illustrates the *preSuperstep* method header.

Listing 2.3: The *preSuperstep* method header.

```
@Override
public void preSuperstep() {
```

The *pre-* and *post-* methods provide partition global bookkeeping but Giraph also has computation global bookkeeping. This is achieved by the usage of the *MasterCompute* class and global aggregators. The aggregator information is shared between the vertices and the master calculation. Changes to this information are available for other vertices after each superstep. The MasterCompute can be used for orchestrating a computation by providing certain values to the aggregators which are later used at partition level. Listing 2.4 provides a MasterCompute class and its methods *compute* and *initialize*, which are used in the graph transformation algorithm. *Compute* is executed one time per superstep at a master worker and *initialize* is executed one time per Giraph computation. The *compute* method maps to the first step of the BSP model.

Listing 2.4: The *MasterCompute* class and some core method headers.

```
public static class MasterCompute extends DefaultMasterCompute {
  @Override
  public void compute() {
  @Override
  public void initialize() throws InstantiationException, IllegalAccessException
      {
```

2.6 Code Generation from Henshin to Giraph

Krause et al. [1] have created and evaluated a code generator where the concept of graph transformations (see Section 2.2), using Henshin (see section 2.3), is mapped to the Bulk Synchronous Parallel model (see Section 2.4), using Apache Giraph (see Section 2.5). This section is dedicated for describing this mapping, the generation of code and the characteristics of the generated code. Their solution has been further developed and integrated into Henshin project since then. Therefore the current Henshin solution is used as a base-point, not the original implementation presented by Krause et al [1].

As explained in Section 2.2, a graph transformation rule application consists of a graph pattern matching phase and a rule application (modification) phase. Using the *CreateCouple* Henshin transformation rule, illustrated by Figure 2.3 on page 12, as an example, the pattern matching phase constitutes finding the the four vertices making up the LHS with their described edges, marked with *«require*»* and *«preserve*»*. The application phase consists of applying the rule on the final match consisting of the two vertices marked *«preserve*»*. Applying the rule constitutes adding the *Couple* vertex and its two edges.

The current implementation of the code generator has some limitations in comparison to the local Henshin interpreter. A rule needs to have exactly one multirule and all the vertices of the LHS need to be contained in it. This is because of the distributed iterative approach enabled by the BSP and to provide the possibility of maximum parallelization at the modelling stage [1]. Further, no attribute conditions can be specified. This is because the current data format only has the possibility to specify the type of the vertex, and no other vertex data, for usage in the computation.

For achieving maximum parallelization Krause et al. [1] used an iterative parallel approach to search the graph for the matches. Since the BSP model (see Section 2.4) consists of a series of supersteps, executed at each vertex, where only local information is available there was a need to iteratively build up the match via local calculations and then send *partial* matches to neighbours. In the following steps the partial matches are extended or deleted to finally apply the rule to the final set of matches. This is done in parallel throughout the whole graph and thereby a lot of these partial matches are passed around.

A (partial) match needs to store information for being able to locate the vertices after the final set of matches is created. The match also should require a minimal amount of memory because of the great amounts of them. Therefore the match consists of a stream of variable length *VertexId*s, where *VertexId* is a unique identifier for each vertex. The vertex id data-type is specified in the *Computation* class header (see Listing 2.1). The order of the ids in the stream represents a position in the LHS of the Henshin rule. The match also needs to store information specific to the match. For example, it is possible to segment the graph, then the segment number of the match is stored in the match. Using the *CreateCouple* and its LHS (represented by Figure 2.6) the partial match will at most consist of four vertices and look like *segmentNumber*:[*p1*, *m1*, *m2*, *p2*], which also represent the order in which the vertices are found.

In order to search the graph and build up a set of matches, a search plan must first be generated. The search plan consists of a series of local steps, *microsteps*, which are performed as separate supersteps of the BSP model. In order to divide the search plan in separate microsteps the Henshin engine analyses the LHS of the rule and outputs a list of steps. These steps constitutes local checks to be performed at each vertex and what to do with the (partial) matches. The search plan is then supplied as an input to the code generator which generates executable code for Giraph. The search plan then consists of a series of *if*-blocks which are executed depending on the

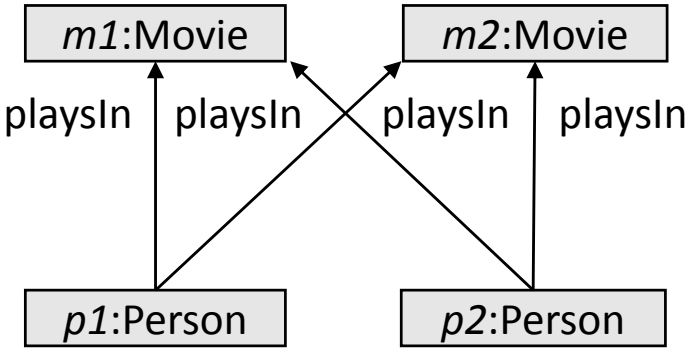


Figure 2.6: A Henshin rule where the LHS and the RHS are equal. The rule illustrates the appearance of a smallest possible target graph where a match for the *CreateCouple* rule can be found.

number of the current superstep. Listing 2.5 shows this for the rule *CreateCouple* which consists of seven microsteps.

Listing 2.5: The structure of the matching code generated.

```
protected void matchCreateCouple(Vertex<VertexId, ByteWritable, ByteWritable>
    vertex, Iterable<Match> matches, int segment, int microstep) throws IOException
     {
    if (microstep == 0) {...
    } else if (microstep == 1) {...
    } else if (microstep == 2) {...
    } else if (microstep == 3) {...
    } else if (microstep == 4) {...
    } else if (microstep == 5) {...
    } else if (microstep == 6) {...
    }
}
```

Since the microstep of the search plan is equal to the current superstep of the overall computation the same microstep is executed for every vertex of the graph. Figure 2.7 illustrated this behaviour for the executing of the 6 microsteps of the *CreateCouple* rule on the source graph in Figure 2.6 on page 18. The microstep executed for the different vertices, represented by vertical arrows, are equal to the superstep of the overall computation. With this behaviour it is extra important with the content of the microsteps which are described below.

The search plan is built up using a series of actions. These actions have the function to iteratively search the graph and build up the match. Often one of these actions are performed per microstep, however they can also be combined. In Subsection 2.6.1 these steps are in depth described using an example. Here follows an introduction: *Local condition* checks are executed in order to determine vertex type, edge type and number of outgoing edges. A partial match can be either *created*, *extended*, *deleted*, *forwarded* without action or two matches can be *merged*. When *extending* or *merging* match(es) the match is checked for injectivity (uniqueness of vertices). In the end of most steps the match is also *sent to* a vertex by using the id as an address. In last microstep the match is *stripped*, *checked for uniqueness* and the rule is *applied*. *Stripping* the match refers to removing vertices not supposed to be a part of the final match.

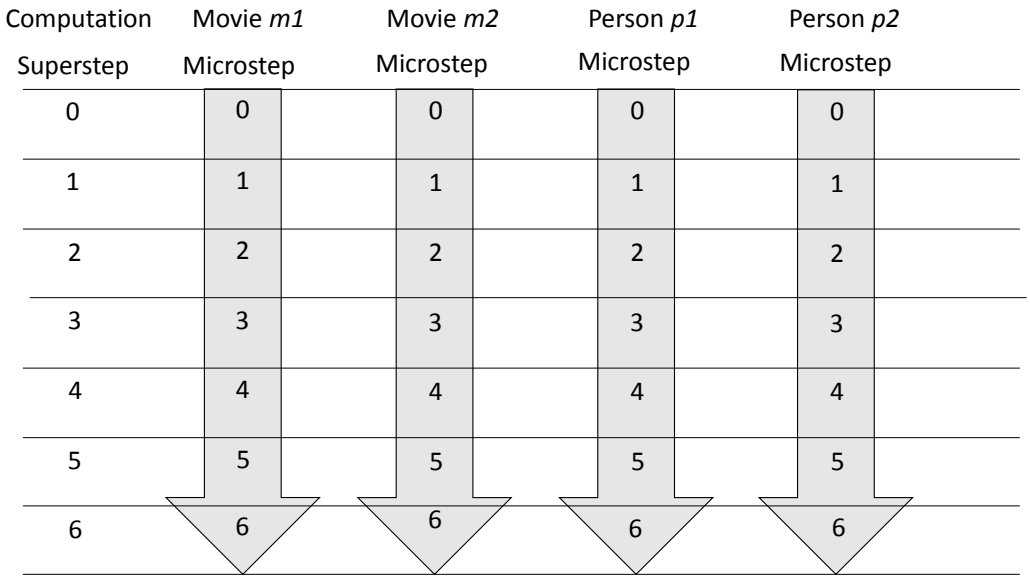


Figure 2.7: The relation between the current superstep and the current microstep for a rule application.

2.6.1 Matching CreateCouple

This subsection aims to exemplify and in-depth go through the search plan and generated matching code for the Rule *CreateCouple*. Figure 2.8 illustrates the matching phase of the execution of the rule on the source graph in Figure 2.6. While the execution of the rule finds two matches and creates several partial matches Figure 2.8 only illustrates the checks resulting in one of these final matches.

In the first microstep (annotated with 0) the *local conditions* are checked. The partial match is *created* and *sent to m1* by *p1*. In microstep 1 *local conditions* are checked and the partial match is *extended* and *sent* back to *p1*. At *p1* microstep 2 is executed by *sending* the partial match to *m2*. In microstep 3 *local conditions* are checked, the partial match is *extended* and sent to *m1*. Next microstep, 4, is a combination of two actions. At *m1* the partial match is *sent* to itself while the second Person is search for by performing local condition checks at *p2*, *creates* a new partial match and *sends* the result to *m1*. In microstep 5 the two partial matches are *merged* to be *sent to p2*. In microstep 6 *p2 local conditions* are compared with the content of the match in order to make sure it has an outgoing edge to *m2*. In the the last microstep the match is *stripped*, *checked for uniqueness* and if all checks succeed the rule is *applied* to the final match.

The code was generated for the rule *CreateCouple* and was executed on the simple graph (Figure 2.6) in order to further clarify the execution and provide a complete picture of it. The result is summarized by Figure 2.1. The column to the far left lists the supersteps of the algorithm. Within such a step a series of matches can be *created* (third column), *extended* (fourth column), *merged* (fifth column), *forwarded* (sixth column) or the rule can be *applied* (seventh column). If nothing is done with the match it is *deleted* (eighth column). The match is sent to the vertex ids

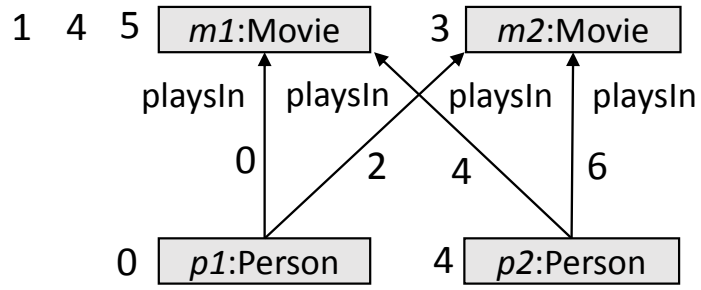


Figure 2.8: Illustrates the order of which the vertices and edges are are found for the rule *CreateCouple*.

in in the column marked *Sends to*. The last column summarizes the number of (partial) matches within the superstep. As can be seen, the characteristics of all the matches (if found) are the same for every vertex except for superstep 4 where two kinds of matches are created. In the last line, the rule is applied after the match is stripped.

Superstep	Vertex	Create	Extend	Merge	Forward	Apply	Delete	Send to	Matches
0	p2	[p2]						m1,m2	
	p1	[p1]						m1,m2	2
1	m2		[p2,m2]					p2	
	m2		[p1,m2]					p1	
	m1		[p2,m1]					p2	
	m1		[p1,m1]					p1	4
2	p2				[p2,m2]			m1,m2	
	p2				[p2,m1]			m1,m2	
	p1				[p1,m1]			m1,m2	
	p1				[p1,m2]			m1,m2	4
3	m1						[p2,m2,m1]		
	m2						[p2,m2,m2]		
	m1						[p2,m1,m1]		
	m2		[p2,m1,m2]					m1	
	m1						[p1,m1,m1]		
	m2		[p1,m1,m2]					m1	
	m1						[p1,m2,m1]		
	m2						[p1,m2,m2]		2
4	p2	[p2]						m1,m2	
	p1	[p1]						m1,m2	
	m1				[p2,m1,m2]			m1	
	m1				[p1,m1,m2]			m1	4
5	m1			[p2,m1,m2,p1]				p1	
	m1			[p1,m1,m2,p2]				p2	
	m2						[p1]		
	m2						[p2]		2
6	p2					[p1, p2]			
	p1					[p2, p1]			2

Table 2.1: Clarifies the execution of the rule *CreateCouple*.

The following text will go even further into the different steps by examining the generated code for *CreateCouple*.

Listing 2.6 shows the code generated for microstep 0. The code is executed for every vertex in the graph independent of type. The objective is to investigate if the current vertex fulfils the requirements of vertex *p1* (see Figure 2.6). Therefore it is first checked whether the vertex is of the required type or subclasses of it (line 1) and the required number of outgoing edges (line 2). If that is fulfilled a new partial match is created with the ID of the current vertex appended to it (line 5). This match is then sent to each adjacent vertex connected with an edge of the correct type (*playsIn*). In order to remove duplicate receivers a *HashSet* is used (lines 7-12). After this step the partial matches existing contains one vertex each.

Listing 2.6: Microstep 0 of *CreateCouple*.

```
1  boolean ok = vertex.getValue().get() == TYPE_PERSON || vertex.getValue().get() ==
       TYPE_ACTOR || vertex.getValue().get() == TYPE_ACTRESS;
2  ok = ok && vertex.getNumEdges() >= 2;
3  ok = ok && (SEGMENT_COUNT == 1 || getSegment(vertex.getId()) == segment);
4  if (ok) {
5    Match match = new Match(segment).append(vertex.getId());
6    matchCount++;
7    Set<VertexId> targets = new HashSet<VertexId>();
8      for (Edge<VertexId, ByteWritable> edge : vertex.getEdges()) {
9        if (edge.getValue().get() == TYPE_PERSON_PLAYS_IN && targets.add(edge.
             getTargetVertexId())) {
10         sendMessage(edge.getTargetVertexId(), match);
11       }
12     }
13 }
```

Listing 2.7 shows the code generated for microstep 1. This step is executed in order to find *Movie m1*. Therefore the type is first checked (line 1). If the type is correct the set of incoming matches, containing *p1*s, is extended and sent back to *p1* (lines 3-10). The *isInjective* methods checks whether a vertex is contained twice in the match. If a vertex is contained more than once (not *injective*) the match is discarded. After this step the partial matches now contain two vertices each.

Listing 2.7: Microstep 1 of *CreateCouple*.

```
1  boolean ok = vertex.getValue().get() == TYPE_MOVIE;
2  if (ok) {
3    for (Match match : matches) {
4      match = match.append(vertex.getId());
5      if (!match.isInjective()) {
6        continue;
7      }
8      matchCount++;
9      sendMessage(match.getVertexId(0), match);
10   }
11 }
```

Listing 2.8 shows the code generated for microstep 2. The previous step only sent messages to vertices of type *Person*, therefore no type checking needs to be performed. The current vertex is not added because it is already contained in the match. Each received match is instead directly sent to adjacent *Movie*s connected via a *playsIn* edge in order to find *m2*, only removing duplicate receivers.

Listing 2.8: Microstep 2 of *CreateCouple*.

```
1  for (Match match : matches) {
2    matchCount++;
3    Set<VertexId> targets = new HashSet<VertexId>();
```

```
4    for (Edge<VertexId, ByteWritable> edge : vertex.getEdges()) {
5      if (edge.getValue().get() == TYPE_PERSON_PLAYS_IN && targets.add(edge.
           getTargetVertexId())) {
6        sendMessage(edge.getTargetVertexId(), match);
7      }
8    }
9  }
```

Listing 2.9 shows the code generated for microstep 3. First, the type is checked (line 1). If the type requirement is fulfilled the incoming matches are extended with the current vertex's id. If the extended match is not injective it is discarded (lines 5-7). This is especially important here because all those previously stored as *m1* are receiving the partial match containing itself. Because the LHS pattern of the rule is regular and the mirror of it gives the same pattern it is discarded if the id of *m1* is less than the current one (lines 8-10). This to remove mirrored matches but duplicate matches. In the end of the step (line 12) the match is sent back to *p1* now containing three vertices.

Listing 2.9: Microstep 3 of *CreateCouple*.

```
1  boolean ok = vertex.getValue().get() == TYPE_MOVIE;
2  if (ok) {
3    for (Match match : matches) {
4      match = match.append(vertex.getId());
5      if (!match.isInjective()) {
6        continue;
7      }
8      if (vertex.getId().compareTo(match.getVertexId(1)) < 0) {
9        continue;
10     }
11     matchCount++;
12     sendMessage(match.getVertexId(1), match);
13   }
14 }
```

Listing 2.10 shows the code generated for microstep 4. This microstep is not dependent on the incoming matches. The incoming matches are instead sent back to the same vertex to be handled in the next microstep (lines 13-19). This will only happen for the vertices of type *Movie*. However, the step serves to find the *Person p2*. Because *p2* is not connected via any incoming edge from the rest of the LHS the step needs to be executed for every vertex of the graph. Therefore, the type is checked and the number of outgoing edges (lines 1-2). In lines 3-12 a new match is created, consisting of only this vertex and sent to all adjacent vertices connected via *playsIn* edge, only removing duplicate receivers. The receivers will be of type *Movie*. After this step it exists two types of partial matches. One containing only one vertex (*p2*) and one containing three vertices (*p1*, *m1* and *m2*). The reason for having to create two matches is that their is no path from *p1* to *p2* or from *p2* to *p1*.

Listing 2.10: Microstep 4 of *CreateCouple*.

```
1  boolean ok = vertex.getValue().get() == TYPE_PERSON || vertex.getValue().get() ==
       TYPE_ACTOR || vertex.getValue().get() == TYPE_ACTRESS;
2  ok = ok && vertex.getNumEdges() >= 2;
3  if (ok) {
4    Match match = new Match(segment).append(vertex.getId());
5    matchCount++;
6    Set<VertexId> targets = new HashSet<VertexId>();
```

```
7   for (Edge<VertexId, ByteWritable> edge : vertex.getEdges()) {
8     if (edge.getValue().get() == TYPE_PERSON_PLAYS_IN && targets.add(edge.
          getTargetVertexId())) {
9       sendMessage(edge.getTargetVertexId(), match);
10    }
11  }
12 }
13 for (Match match : matches) {
14   VertexId id = match.getVertexId(1);
15   if (vertex.getId().equals(id)) {
16     matchCount++;
17     sendMessage(id, match);
18   }
19 }
```

Listing 2.11 shows the code generated for microstep 5. This step is executed for every vertex, however only vertices of type *Movie* will receive any messages and therefore perform any actions. The purpose of the step is to join the two different kinds of matches now existing. In lines 3-10 they are added to two different buckets depending on what type of match they are. Either they contain three ids or one. In lines 11-20 all the combinations of matches from the two buckets are merged together, checked for injectivity and sent to *p2*. All the matches now are of size four.

Listing 2.11: Microstep 5 of *CreateCouple*.

```
1  List<Match> matches1 = new ArrayList<Match>();
2  List<Match> matches2 = new ArrayList<Match>();
3  VertexId id = vertex.getId();
4  for (Match match : matches) {
5    if (id.equals(match.getVertexId(1))) {
6      matches1.add(match.copy());
7    } else {
8      matches2.add(match.copy());
9    }
10 }
11 for (Match m1 : matches1) {
12   for (Match m2 : matches2) {
13     Match match = m1.append(m2);
14     if (!match.isInjective()) {
15       continue;
16     }
17     matchCount++;
18     sendMessage(match.getVertexId(3), match);
19   }
20 }
```

The objective of microstep 6, which is shown in Listing 2.12, is to check that *Person p2* has an edge to the vertex registered as *Movie m2*. This is done via iterating over all the incoming matches (line 1) the outgoing edges (line 3) and removing duplicates. For each outgoing edge of type *playsIn* pointing at a unique vertex, the Movies are removed from the match and the rule is applied on the match (line 10). The *if*-statement will always be true because in this example no segmentation is used. If segmentation is used, the vertex forwards the message for the rule to be applied when the rule is executed for the last segment. The reason for removing the two *Movies* from the match before creating the final set of matches is because they in the rule are marked with *«require*»* and should therefore not be considered a part of the final match (see Section 2.3).

Listing 2.12: Microstep 6 of *CreateCouple*.

```
1  for (Match match : matches) {
2    VertexId targetId = match.getVertexId(2);
3    for (Edge<VertexId, ByteWritable> edge : vertex.getEdges()) {
4      if (edge.getValue().get() == TYPE_PERSON_PLAYS_IN && edge.getTargetVertexId()
           .equals(targetId)) {
5        match = match.remove(2);
6        match = match.remove(1);
7        if (finalMatches.add(match)) {
8          matchCount++;
9          if (segment == SEGMENT_COUNT - 1) {
10           applyCreateCouple(vertex, match, appCount++);
11         } else {
12           sendMessage(vertex.getId(), match);
13         }
14       }
15       break;
16     }
17   }
18 }
```

The manipulation of the graph, for example performed by the method *applyCreateCouple* in Listing 2.12, is by Krause et al. [1] considered less performance intensive than the matching phase and is therefore not considered further here. However the application of the rule only takes one superstep and is thereby included in the last superstep of the matching.

The example provided above may be seen as simple, however the code generator must handle arbitrary rules and is therefore rather complex. It must also handle different kinds of operational unit. For example the sequential unit of Figure 2.4 must be correctly scheduled. Conceptually this is done via letting the *MasterCompute*, which corresponds to step 1 of the BSP model (see Section 2.4), schedule the different rules and make the information about current rule and microstep available for the rest of the Giraph partitions.

Listing 2.4 on page 16 presented the *MasterCompute* class and its methods *compute* and *initialize*. These methods are used to schedule the rule and unit execution. Listing 2.13 lists the code executed when *initialize* is called. Initialize is called one time per calculation and sets up four aggregators. These aggregators are available from all workers in the computation. Most interesting is the *AGGREGATOR_APPLICATION_STACK* which holds a stack over the last applied rule or unit.

Listing 2.13: The code in *initialize* method.

```
1       registerAggregator(AGGREGATOR_MATCHES, LongSumAggregator.class);
2       registerAggregator(AGGREGATOR_RULE_APPLICATIONS, LongSumAggregator.class);
3       registerPersistentAggregator(AGGREGATOR_NODE_GENERATION, LongSumAggregator.
            class);
4       registerPersistentAggregator(AGGREGATOR_APPLICATION_STACK,
            ApplicationStackAggregator.class);
```

Listing 2.4 shows the content of the *compute* method which is responsible for scheduling. This code is executed in the beginning of every superstep (step 1 of the BSP model). If this is the first superstep of the calculation a new stack is created and the first rule or unit to be executed is appended to the stack (lines 3-4). If it is not the first superstep the current stack is received from the aggregator. In both cases

nextRuleStep is called for further scheduling (lines 5 and 8) and the current stack is saved using the aggregator (line 10).

Listing 2.14: The code in *compute* method.

```
1       ApplicationStack stack;
2       if (getSuperstep() == 0) {
3         stack = new ApplicationStack();
4         stack = stack.append(UNIT_ADD_COUPLES, 0, 0);
5         stack = nextRuleStep(stack, ruleApps);
6       } else {
7         stack = getAggregatedValue(AGGREGATOR_APPLICATION_STACK);
8         stack = nextRuleStep(stack, ruleApps);
9       }
10      setAggregatedValue(AGGREGATOR_APPLICATION_STACK, stack);
```

Listing 2.15 shows the method *nextRuleStep*. The current unit (can also be a rule), segment and stack are popped from the stack (lines 4-7). Depending on the content of *unit* different methods are called for scheduling the individual rules. In the end of the *nextRuleStep* method it is returned if the current stack holds a declarative rule (lines 21-29).

Listing 2.15: The method *nextRuleStep*

```
1     private ApplicationStack nextRuleStep(
2       ApplicationStack stack, long ruleApps) {
3       while (stack.getStackSize() > 0) {
4         int unit = stack.getLastUnit();
5         int segment = stack.getLastSegment();
6         int microstep = stack.getLastMicrostep();
7         stack = stack.removeLast();
8         switch (unit) {
9         case UNIT_ADD_COUPLES:
10          stack = processAddCouples(stack, microstep);
11          break;
12        case RULE_CREATE_COUPLE:
13          stack = processCreateCouple(stack, segment, microstep, ruleApps);
14          break;
15        case RULE_CREATE_OCCURRENCE:
16          stack = processCreateOccurrence(stack, segment, microstep, ruleApps);
17          break;
18        default:
19          throw new RuntimeException("Unknown unit " + unit);
20        }
21        if (stack.getStackSize() > 0) {
22          unit = stack.getLastUnit();
23          if (unit == RULE_CREATE_COUPLE ||
24            unit == RULE_CREATE_OCCURRENCE) {
25            break;
26          }
27        }
28      }
29      return stack;
30    }
```

The three methods called *process** behaves differently depending on if it is a rule or a unit. For a rule it is simply counting up the microstep until all microsteps of that rule are executed to then repeat for every segment of the graph. For a unit it behaves differently for different kinds of units. However the new implementation do not affect these methods and therefore they are not described in detail.

Listing 2.3 presented the *preSuperstep* method. The method is working as a bridging coordination between the *MasterCompute* and the rule application at each vertex in

the graph. It is executed one time per partition and makes information about rule, microstep and segment locally available.

3

Method

3.1 The Design Methods and Procedures

The research will be conducted in the form of design science and follow the framework with the seven principles presented by Hevner et al. [18]. The rest of this section presents the seven principles and how this research addresses them. The research will attempt to answer the in Section 3.2 presented research questions.

The first guideline, *"Design as an Artifact"* [18], will be approached by the production of a viable instantiation artifact. The instantiation will be in the form of a software tool which aims to improve the by Krause et al. [1] already constructed tool with respect to trying to give a positive answer to research question RQ2. The software tool will be based on the technology presented in Chapter 2. The design also has to take into consideration the basic rules of parallel computing, the ACID (Atomic, Consistent, Isolated and Durable) properties.

To address the second guideline, *"Problem relevance"* [18], the work will develop a solution which is aimed to solve the relevant business problem stated in Chapter 1. The solution will attempt to answer the for the guideline relevant research question, RQ1.

Further, guideline 3, *"Design evaluation"* [18], will be fulfilled by the rigorous demonstration of the instantiation's utility, quality and efficacy using purposefully designed artificial benchmarks in the form of synthetic graphs. Dynamic analysis (the artificial benchmarks) will evaluate for the dynamic qualities asked for by research questions RQ2 and RQ3.

Also a real data set will be used. The choice for the real data set is the one used by Krause et al. [1]. The IMDB movie database. Then the solution will be possible to more easily evaluate in relation to the already available results. It will also be beneficial to use a real data set to evaluate the benefit the instantiation has on real data.

On the other hand, the set of artificial benchmarks will be purposefully designed for use to evaluate on which cases the solution performs better and on which cases (e.g. sparse vs dense graphs) it will have worse performance. Thus for aiming to respond to RQ3. It will also be possible to relate the different solutions to each other and possibly come up with a hybrid solution. For more information about

the benchmarks see Chapter 6 and for more information about the evaluation see Chapter 8.

The fourth guideline, *"Research Contribution"* [18], is going to be achieved by making the contribution clear and verifiable in the area of the design instantiation (graph transformations and the BSP model) and the design foundation by aiming to improve and extend the available knowledge.

By applying rigorous methods for the construction of the instantiation and when designing the methods for evaluation the fifth guideline, *"Research rigor"*, shall be taken into consideration. The steps undertaken shall be explicitly validated, using the above described evaluation plan, and purposefully described.

Guideline 6, *"Design as search process"* [18], is going to be expressed in the way of conducting the research. The research will be conducted in an iterative manner using the generate/test cycle presented by Hevner et al. [18]. During the generate phase the research question RQ1 will be attempted to answered and during the test cycle research question RQ2 and RQ3 will be answered utilizing the described evaluation approaches. The result will serve as feedback for the next cycle.

The last guideline, *"Communication of research"* [18], is going to be followed by apart from describing the solution in a technical manner for practitioners and researcher within the technology also describe the solution with business aim. This will according to Hevner et al. [18] enable behaviour science to research the use of the instantiation within an organization.

3.2 Research Questions

Using the in Section 3.1 described method, the following research questions will be addressed:

> RQ1: How can the performance and memory consumption of the current approach be improved?

For each alternative answering RQ1:

> RQ2: Are the performance and memory consumption improved by the alternative solution?
>
> RQ3: Which properties of the source graph and the graph transformation rule is the alternative suitable for?

3.3 Limitations and Delimitations

The concepts of the underlying solution will not be limited to the technical environment (EMF, Henshin and Giraph), however it will not be evaluated using other technical platforms. This is enabled by focusing on the theoretical concepts of

Bulk Synchronous Parallel and graph transformations. The reason for not evaluating using other platforms is because the implementation is time consuming. Also the evaluation (or benchmarking) is heavy on resources and requires substantial time.

4

Review of the Literature

4.1 Foundation

Arendt et al. [6] describes Henshin which is a part of the Eclipse Modelling Framework (EMF). The features of Henshin is in depth presented using the case of model refactoring and meta-model evolution. As Henshin is used as the graph transformation language and plugin in this thesis, their paper has been essential in order to understand, develop and use it. They also provide a more general theory oriented-presentation than the available documentation.

Krause et al. [1] use graph transformation rules and units as a high level modelling language containing operational and declarative features. To be able to apply the language on large scale graphs, a mapping to the Bulk Synchronous Parallel model is introduced. The approach is highly distributed and parallelized. The solution is evaluated with a large dataset using a large cluster. The tools used are the Henshin modelling tool and the Apache Giraph framework on top of Hadoop. This paper lays the foundation for the knowledge this thesis will build on as well as suggesting the future research which this thesis will conduct.

Guo at al [19] has performed a benchmark of different graph processing platforms of which Giraph is one of them. Although this thesis is not intended to compare different platforms but rather different ways of implementing algorithms within the same platform many ideas can be borrowed. For example, dataset diversity is identified together with platform and algorithm diversity. Here the notion of dataset diversity and algorithm diversity can be used in the for this thesis created benchmark suite.

On the other hand although their paper describes performance benchmarks on the Giraph platform with a variation of algorithms and data-sets their benchmark results cannot be used as a foundation for this work. This because they use real data with rather unclear characteristics and a set of algorithms with the main variation points of the number of iterations and the amount of processing. From their results it is therefore unclear how a data-set with certain characteristics will perform as well as how a certain change to the algorithm will impact performance.

Guo et al. [19] further identify raw performance, resource utilization and scalability to be evaluated. Those measurements provide inspiration for what is going to be

measured in the evaluation. Their paper also provides valuable insights related to the underlying computation platform, Hadoop. Their study shows that YARN, which is a modern implementation of MapReduce, would have better performance than MapReduce while running graph processing algorithms directly on it. Since Hadoop (with MapReduce) is the base for Giraph it would be interesting to see if a switch to YARN would provide a significant improvement. However it is out of scope of this work but is suggested for future work.

4.2 Related Work

Izso et al. [8] have identified a problem with the scalability of current Model-driven engineering (MDE) tools. They have also identified solutions in other fields, although those solutions create more complex and low-level queries. To address the problem of scalable MDE tools while preserving high abstraction they adapt techniques for incremental graph search. The techniques are known from the project EMF-IncQuery[20] and adapted for a distributed cloud infrastructure. The solution is claimed to be highly scalable. The paper provides another approach for doing the study suggested. Although the authors claim the solution to be horizontal scalable, to work on very large models and efficiently handle complex queries it is unclear how it scales for rules with a high amount of partial matches as in the example handled by Krause et al. [1] and which this thesis builds on.

Bergmann et al. [21] present an incremental pattern matching technique for graph transformations. The approach stores all matches as a cache which is updated upon model changes. The approach sacrifices model manipulation performance and memory consumption for faster pattern queries. Possibilities for parallelization are also discussed. The paper can offer ideas regarding performance versus memory usage and similar considerations. However large scale graphs distributed over a cluster is not considered and it also uses a high amount of memory.

In another article Bergmann el al [22] describes the evaluation of incremental pattern matching for graph transformations using a benchmark. The approach has similar characteristics as the one presented in [21]. A cache for all the matches is maintained while the model evolves. This approach sacrifices flexibility and memory consumption in comparison to the solution provided in this thesis. It also requires the model to be constant in memory and not in the form of a batch computation as the one provided in this work. The paper also illustrated the benefits of their solution in similar manner as supplied in this thesis.

5

Qualitative Analysis on Performance Issues

In order to answer research question RQ1 (see Chapter 3) there is a need to implement alternative solutions and evaluate them. This chapter aims to present the initial analyses which led to the decision to implement and evaluate the alternative solution. The foundations presented in Chapter 2 are used as a base-point for the analysis.

The problems could depend on many factors. However, in order to discuss and also be able to evaluate the discussion is limited to the following: First, possible problems with the usage of the BSP model is discussed in Section 5.1. Secondly, potential problems with the mapping of graph transformations to the BSP model is discussed in Section 5.2. Section 5.3 provides discussion of what happens when the cluster, which executes the computation, is scaled out. And lastly, possible issues with the current generated code is discussed is discussed in Section 5.4.

5.1 Bulk Synchronous Parallel

As illustrated by Figure 2.5 on page 14, the Bulk Synchronous Parallel model (see Section 2.4) is massively parallel. Thereby, algorithms implementing it have the opportunity to take full advantage of the parallelism. The parallelism in combination with the simplicity is the main advantages of the model. However, the simple and rigid structure with all vertices computing (Step 2 - Vertex Computation) at the same time and all vertices sending messages (Step 3 - Vertex Communication) at the same time can cause bottlenecks.

The cause of the bottlenecks is the concentration of particular resource requirements to certain points in time. The resource requirements of particular interest are memory, CPU time and bandwidth. Further, the fourth step (Synchronization) requires all vertices to be done computing and sending messages before continuing the computation which makes the computation time dependent on the slowest vertex computation and compute node which can leave resources not fully utilized while waiting for synchronization.

In the vertex computation phase, vertices compute, create local variables and store

incoming and outgoing messages. Therefore the speed of the phase is dependent on the main resources; the usage of CPU time and memory. The phase can be expected to require an amount of memory which is dependent on the number of active vertices in a superstep, their usage of local variables and their storage of messages. CPU time used will be dependent on the number of nodes, the number of incoming messages to process and how complex the calculation is. The most limited resource will most likely determine the time.

In the next phase, vertex communication phase, the required bandwidth is expected to be dependent on the number of messages sent in each superstep and the size of them. Reducing the amount of messages therefore result in improvement of both bandwidth usage and memory usage. The reason is that the vertices communicate via message passing locally, at thread and worker level, resulting in memory consumption. They also communicate between nodes which is resulting in the usage of the network bandwidth. Therefore, one should generally try to minimize the resource requirements or use the one with less constraints on, e.g. memory versus CPU time. Another possibility is to distribute the requirements across supersteps in order to even out the different requirements.

5.2 Mapping Graph Transformations to BSP

The mapping of graph transformations (see Section 2.2) to the BSP model and the code generator (see Section 2.6) fully utilizes the parallelism in the BSP model. It is therefore sensible to the in Section 5.1 presented shortcomings of the general BSP model. However, the solution also has possible problems specific for the algorithm.

Matches are iteratively searched for and built up to lastly apply the rule to all matches in parallel. However, as presented in Section 2.6 and as illustrated by Figure 2.7 on page 19 the number of the microstep executed equals the number of the overall superstep. This implies that every vertex of the graph is attempting to execute the same snippet of code at the same point. Therefore all vertices are according to the discussion about potential problems with the BSP model (see Section 5.1) having approximately equal amount and and type of resource requirement at the same superstep. And this continues for every superstep in the computation.

All vertices also follow the same iterative search plan (see Listing 2.5 on page 2.5). This means that the vertices also create the same type, amount and sizes of partial matches across the graph. In step 3 of the BSP algorithm (see Section 2.4) all the messages are sent concurrently. Recap Table 2.1 on page 20 and imagine what will happen if the source graph is made considerably larger. For example, superstep 5 will then have to handle an even larger amounts of incoming messages while others like superstep 0 will create large amounts of small matches. This will have different impacts for different microsteps.

To further illustrate this, the rule *CreateCouple* (see Figure 2.3 on page 12) was executed on a million of vertices large graph using 24 compute nodes (see Chapter 6

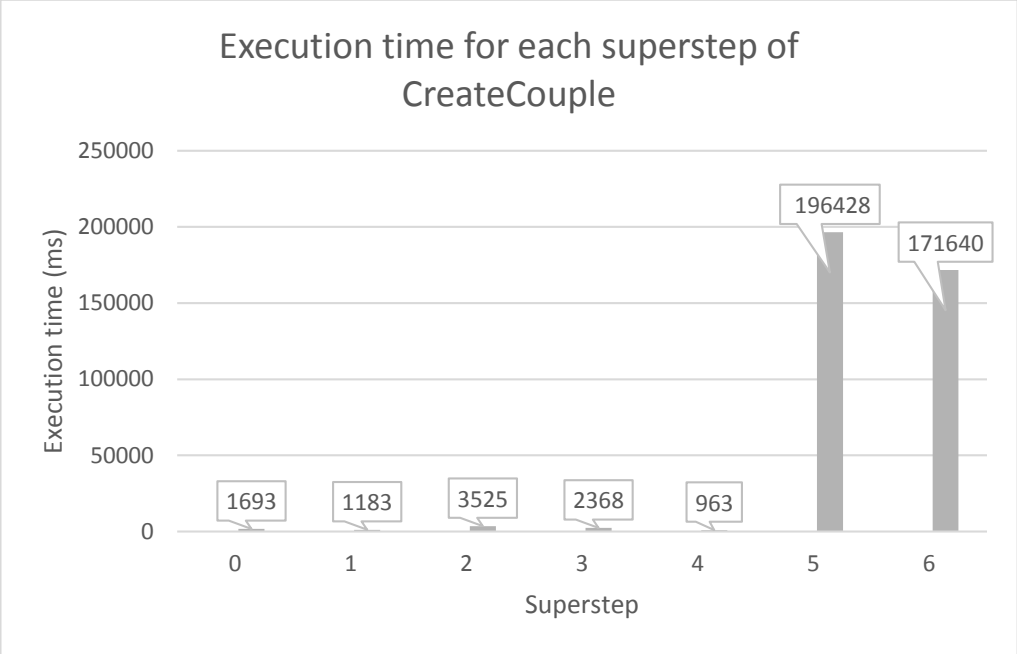


Figure 5.1: The figure shows the execution time per superstep for the rule *CreateCouple* averaged over three executions on 24 nodes.

for more detailed configuration). The amount of used time per superstep was gathered. Figure 5.1 provides a column chart for the execution. Three executions were run and the average is used here. It is rather clear that microstep 5 and 6 require considerable more time than the rest. However, there are also large differences when superstep 2 and 3 are compared to the rest of the supersteps.

The result in the figure is also reasonable if we study the microsteps in more depth by looking at the code (see Listings 2.6-2.12 on page 21 to 23). See for example microstep 2 (Listing 2.8 on page 21), which simply passes the received matches forward, compared to microstep 5 (see Listing 2.11 on page 23), which copies the two matches into two lists, performs a join of two entries in the two lists and checks the result for injectivity. The *forward*-step requires small amount of both memory and CPU time for the computation while the *join-step* has much heavier requirements on both. For example the microstep 5 quickly doubles the memory used for the partial matches. And the second loop, which used the matches, has a quadratic complexity. Now also taking into consideration executing this on millions of vertices large graphs this might lead to considerable amounts of garbage collection work and compute requirements within step 2 of the BSP superstep leading to slow computations for the most performance intensive microsteps which is confirmed by the figure.

Microstep 6, which according to Figure 5.1 also takes considerable time, is also interesting to study. However, the step contains no actions which look to be performance intensive apart from applying the rule on the matches (see Listing 2.12 on page 23). Applying a rule on a match leaves little or no room for improvements for the Giraph-user and is therefore not further considered.

Another problem with the mapping to the BSP model is the lack of parallelization

in the iterative matching build up. If the LHS consists of several, from each other unreachable, vertices the matching takes more supersteps than what is actually needed using a slightly different approach. See for example the LHS of *CreateCouple* which has two vertices of type *Person* unreachable from each other. This results is that a new match is created in microstep 4 (see Listing 2.11) while the already created one is passed forward without action. If further parallelization of this could be achieved; the result would be a shorter execution in terms of supersteps.

Further, many messages are sent to only in the next step being quickly deleted. This because the matches fail a simple check like the check for injectivity. This irrespective of that the information required to delete it already was available in the previous step. In Table 2.1 on page 20 an example of this is provided between superstep two and three were for example the match [*p1,m2*] is sent to *m2* only to be deleted. However, this is another downside with using the BSP model and cannot be solved for most cases. Vertices only have local information available. The current vertex only knows the ID of the receiving vertex. This happen because the BSP itself build on the idea that vertices should only have local information.

5.3 Horizontal Scaling

As shown in Figure 1.1 on page 3 the improvement when adding more compute nodes (scale out) starts decreasing drastically at a certain number of nodes. When the number of nodes used for a Giraph (see Section 2.5) computation is increased the source graph will be partitioned in even more partitions. This according to the above discussion about resource requirements (see Section 5.1) would result in faster execution because of the extra memory and cores made available via the extra node. However, more partitions also results in considerable more (partial) matches sent across nodes, which will consume network bandwidth. Assuming that the bandwidth is constant when adding more nodes this can cause a bottleneck. Therefore a possible path forward is to reduce the sizes and amounts of (partial) matches passed around at particular steps. Another is to control the partitioning.

When combining the discussion about more partitions and the, in Section 5.2, discussion about different resource requirements for different microsteps it is possible that the peaks are made even higher and therefore causes the bad scaling. To investigate if this is the problem the benchmarking suite of Chapter 6 is created with the goal of sending as much messages as possible. However, since Figure 5.1 shows that microstep 2 takes little time in comparison to the two last steps it seems unreasonable that this is a main concern.

5.4 Issues with the Current Generated Code

This section describes the specific implementation issues found with the code generated for Giraph. It involves rather more details than previous sections. Subsec-

tion 5.4.1 provides a discussion about the choice of the data type for a match and how that might affect performance. Subsection 5.4.2 discusses more scattered issues about the code.

5.4.1 The Match

The previous sections of this chapter have pointed out that the messages sent are important and therefore they are here further discussed with a focus on the implementation of the data type. In Section 2.6, the format for storing the partial matches, the Match, was presented. In short it consists of a stream with variable sized id's of the vertices. Since the id's themselves are variable sized and they are stored in a stream the match requires as little memory as possible. This makes the messages light to pass around and this is beneficial because of the great numbers of them required for large graphs.

However, making the match as small as possible sacrificed CPU time for reducing the amount of memory and bandwidth used. Usually simple operations such as getting the nth element are no longer in the magnitude of O(1) in complexity. This because of the size of the vertices are unequal and therefore the *n-1* first vertex ids needs to be checked for their length in order to access number n. This can for example be seen inside two nested loops in the performance intensive fifth microstep of the *CreateCouple* (see Listing 2.11 line 18 on page 23). Further, this also affects other operations such as *remove* and *isInjective*.

Another downside with the minimal matches is when checking for injectivity. Injectivity needs to be checked whenever a new id of a Vertex is added to the match and involves checking for duplicates. This is performed via creating objects for all VertexIds in the match, and adding them to a HashSet to find duplicates. Since the match cannot contain duplicates the natural choice would have been making it a set or an ordered set to also maintain the order. Since the current approach needs to create several temporary used object, apart from the already stored match, it also uses significant memory during the computation. However, the algorithm uses the position in the match for further identification of the position of vertices within the rule. This means that an alternative solution has to store the order in some other way.

5.4.2 Other Issues

As can be seen throughout the for the rule *CreateCouple* generated code, many objects are created throughout the code. Many of them are created unduly. For example. In microstep 5 (see Listing 2.11 on page 23) some matches are appended to each other only to be deleted due to the injectivity check. In microstep 6 (see Listing 2.12 on page 23) two consecutive nodes are deleted in separate operations were the sub-result is of no interest. Further, other operations as for example the injective check creates objects for all vertex ids to simply delete them.

It is apparent considering the discussion in Section 5.2 that improvements of microstep 5 and 6 have the most impact on the overall computation. Since it is hard to find improvements of microstep 6 according to the previous discussion microstep 5 are given the most attention. Both memory usage and CPU time looks to be heavy. If one can improve that either by reducing the need for copying matches, the number of laps in the nested loop or the content of the loop; the large improvements could possibly be gained.

6

Benchmarking Methods and Procedures

This chapter aims to define and present the procedure used for the evaluation presented in Chapter 8. Firstly, Section 6.1 aims to present the variation sources and constants, while Section 6.2 presents the metrics used. In Chapter 3, the idea of creating a benchmark suite containing synthetic graphs with the purpose of comparing the different implementations was presented. In Section 6.3, the synthetic graphs are presented and derived using the notion of graph metrics as presented by Berge [23]. The section also contains information about a real data example used. Finally, Section 6.4 presents the graph transformation rules to execute on the source graphs presented.

6.1 Variation Sources and Constants

Guo et al. [19] have identified three sources of variation for a benchmark. These are platform, algorithm and dataset diversity. Platform in this case will be Java, Giraph and Hadoop. Since the goal is not to optimize the platform it is set constant. Algorithm is the variation point which is going to be optimized and therefore set as the independent variable. In order to do so a set of datasets needs to be created and used as another independent variable.

The first constant is the platform. To measure the performance of the actual code generated and not random events related to hardware and software environment extra considerations were placed on the hardware and software configuration. Since purpose of this thesis is to improve the performance of the code-generator which generates Java code for Giraph to run on Hadoop this thesis must also use the same platform. Therefore the versions: Java 1.7.0_79, Giraph version 1.1.0 and Hadoop version 0.20.203 was used. The Henshin version used as a base-point is the head revision at 2015-02-20. Rest of the software and hardware environment was provided by the Glenn cluster at Chalmers University of Technology in Gothenburg, Sweden. The machines in the cluster are based on AMD Opteron 6220 with a total of 16 cores per machine, clocked at 3 GHz, and 32GB RAM per node. The communication is handled via Infiband based on the Mellanox ConnectX-2 QDR Infiniband 40 Gbps HCA's.

There are also specific configurations of Giraph which are fixed over the experiments. The number of Giraph workers are equal to the amount of cluster nodes utilized for a computation. The number of threads used per worker is 16. In this case this means 16 threads per node. The number of partitions the source graph is divided in is equal to the number of threads per machine times the total number of workers. This means one partition per thread. The number of threads used to write and read the data is one. Finally, the amount of memory used per node equals 28 GB if nothing else is mentioned.

However, there are potential problems with the cluster setup. The first problem is that other jobs are executed concurrently on other nodes. These jobs also cause network traffic which can cause variations in the available network bandwidth. Lastly, the allocation of nodes are not controlled. One job can get two adjacent nodes and and another job can get two nodes far apart which impact the communication capability between the nodes.

The first variation source for this benchmark is the set of the synthetic graphs and real data example to use as source graphs for the graph transformation executions. Since the thesis aims to investigate which factors are limiting the performance in order to answer research question RQ1 (see Section 3.2) and what properties of the source graph the alternative implementation is especially suitable for in order to answer research question RQ3, the notion of graph metrics was used to derive the graphs. In order to provide an exhaustive set of synthetic benchmarks the notion of graph metrics is used for the creation of a complete set of synthetic graphs for benchmarking purpose. The basic idea is that the different synthetic graphs should have high respective low values of the different metrics. This will enable the testing of edge cases and help to see in what cases a certain implementation performs better or worse as well as ultimately finding out the core problem. More information about this is provided in Section 6.3.

The second variation source is the alternative implementations of the code generator. These are presented in great detail in Chapter 7. In order to try the different implementations, transformation rules needs to be created in order to generate code from. These rules need to be complete; contain all the actions presented in Section 2.6. They also needs to be designed in order to answer research question RQ3. Section 6.4 describes them in greater detail.

6.2 Metrics

In order to evaluate the performance of the different algorithms and thereby answer research question RQ2 (see Chapter 3) a definition of what constitutes superior performance must be made. To do so the performance aspects of interest are first identified. The metrics are then designed to measure the aspects. The metrics are presented in detail using the ISO 15939 [24] method. The following paragraphs will introduce the metrics via linking and explaining the need for the metric and present name, unit, formal definition and method of collection in tabular form.

Table 6.1: Metric 1.

Name:	Execution time as a function of number of nodes.
Unit:	ms
Formal definition:	$f(n)$ where n is the number of nodes
Method of collection:	Time outputted from a Giraph job.

Table 6.2: Metric 2.

Name:	Total used node time.
Unit:	ms
Formal definition:	$f(n) * n$ where n is number of nodes
Method of collection:	Time outputted from a Giraph job.

Guo et al. [19] identified for the purpose of their evaluation four performance aspects of interest. Of these, three have been used for deriving the aspects used in this this work. These are processing speed, memory consumption and scalability. They also identified overhead. However, that is not chosen since it is mainly dependent on the platform and the aim is not to compare platforms as in their evaluation.

The initial indicator of the performance problems was time as a function of number of nodes (see Chapter 1). The data was provided by Krause et al. [1] in their evaluation and presented in Figure 1.1. Since they used time as a function of number of nodes and total used node time it is natural to select those. This because the results should be easy to compare and relate to previous result, but also that they provide suitable metrics. Time as a function of number of nodes can also be called *scalability* and the overall purpose of this thesis (see Section 1.3) was to improve the horizontal scalability. Further, adding nodes should reduce time used however not increase the time used per node significantly. When comparing these measures the scalability can be evaluated. Table 6.1 and 6.2 provide detailed information about the metrics.

As indicated by the problem analysis (see Chapter 5) memory consumption and network bandwidth can possibly be the limiting factors and therefore they need to be measured. However, the cluster used for the evaluation do not allow exclusive execution rights. Other jobs are executed in the mean time on other nodes and therefore causes network traffic. Therefore, network bandwidth is not measured because the metric would be too unreliable.

However, memory usage is measured in order to find out if the alternative solutions uses less or more memory in comparison to the old solution. While memory consumption is of less importance as an end result (because a whole node is allocated for each job) it is of importance as an identifier in order to find the source of the problem. Bergmann et al. [21] for example describes the need for sacrificing memory consumption for increased execution speed. It is measured using a series of executions of a transformation rule on a pseudo-distributed (1 node) cluster. In this

Table 6.3: Metric 3.

Name:	Memory test
Unit:	MB
Formal definition:	$t = g(m)$ where t is time and m is memory
Method of collection:	Binary search to find the least amount of memory resulting in successful execution.

particular case the memory is not constant anymore, as was explained in Section 6.1, but the rest of the platform still is constant. The memory is instead reduced gradually until the execution fails or a predefined time-limit is triggered. The reason for the time-limit is that the time used for the execution will increase significantly when the memory limit is approaching. This because of garbage collection. There is also a usage limit for the cluster and executing long jobs will quickly drain this project of resources. Table 6.3 provides detailed information about the metric.

6.3 Source Graphs

The evaluation is performed using a set of synthetic graphs, designed for the specific purpose of the evaluation, and a real data example as source graphs. This section aims to present these. The synthetic graphs are derived using graph metrics to achieve a comprehensive set.

The real data example is borrowed from the work of Krause et al [1]. It is a part of the IMDb movie database dated July, 26th 2013. The data adheres to the metamodel illustrated by Figure 2.1 and contains 924 054 movies, 1 777 656 actors and 980 396 actresses. The source data contains no couples since they should later be created (see Section 6.4).

The synthetic graphs are derived using graph metrics. It exists two types of graph metrics: local and global metrics, where the latter is of the most importance here since the whole graph needs to be considered to control overall memory and processing power usage. Local variations can be disregarded. Global graph metrics can be simple metrics such as the number of vertices and the number of edges, but also more complex ones. Because this is a first attempt to optimize for different graph characteristics an overweight of simple metrics has been chosen and is dealt to be sufficient at this stage. Further, only graphs which are fully connected are considered. Otherwise they are considered as separate graphs. The following metrics is chosen:

- VertexCount - The number of vertices.
- EdgeCount - The number of edges.
- Degree - The number of edges per vertex on average.

Above, the *VertexCount* and *EdgeCount* are simple measures of size while *Degree* tells more about the structure of the graph. The basic idea is of course that the total number of vertices and edges has a linear effect on the execution time, but more importantly are the other measures. The *Edge-* and *VertexCount* can also be called tightness. *Degree* is a measure of the number of in- and outgoing edges. If the value is high the graph is said to be highly connected and if it is low it is sparse.

Recall the matching algorithm presented in Section 2.6. The algorithm iteratively builds up the match at the vertices by checking local connections and sending partial messages to neighboring vertices. Therefore, executing the algorithm on a more highly connected graph more messages will be sent. More memory is then consumed for a more highly connected graph. For example a graph which looks like a line, with one or two edges per vertex, would be an extreme example. Another would be a graph with one central vertex and all existing edges are from this node to other vertices.

Kuo et al. [25] has collected and used a set of simple graphs which for example Rauzy [26] also uses in an evaluation. The set of graphs are presented in Figure 6.1. The graphs were not created for big data and are therefore not suitable for scaling and includes graphs that would not make sense in such a case. However, it worked as a source of inspiration for the development of the synthetic graphs used for this thesis. Graphs with different properties may perform differently well with different transformation algorithms. Especially graphs (27) and (28) in the picture, which are similar to the graphs used in this thesis, inspired.

Figure 6.2 illustrates the designed graphs. They constitute two types of graphs with two points of variation each. First, Sub-figure 6.2b illustrates the *Net*. This graph is a regularly formed grid with edges only in one direction. The edges points from lower left corner upwards and to the right. The other form of graph, as illustrated by Sub-figure 6.2d, is the *rope* which is a regularly formed structure with every vertex having exactly one edge (except the last one) to the next vertex. The points of variation are the sizes (millions of vertices) and bidirectional edges or not. Sub-figure 6.2a and 6.2c illustrates the bidirectional alternatives. Since bidirectional edges is not a function of Henshin (see Section 2.3) this is created with edges in both directions. These graphs are simple, however they provide enough variation. The number of vertices can be varied, the number of edges as well so as the degree which is either one or two in the unidirectional case or 3 or 4 in the bidirectional. Thus, the number of edges are doubled in the bidirectional cases. The point is that when the number of edges increase more partial matches (see Section 2.6) will be sent and this will result in a higher memory usage .

These graphs have similar structure and therefore share a common meta-model. The meta-model is presented in Figure 6.3. This meta-model consists of two different kinds of vertices, a *VertexContainer* and a Vertex. The VertexContainer is supposed to be one per graph and has the purpose of being a parent to all vertices via the composition association named *vertices*. A vertex of the type *Vertex* has four types of edges to other vertices. They are named *east*, *west*, *north* and *south* These are used for easier generation of the graphs compared to having only one type of edges.

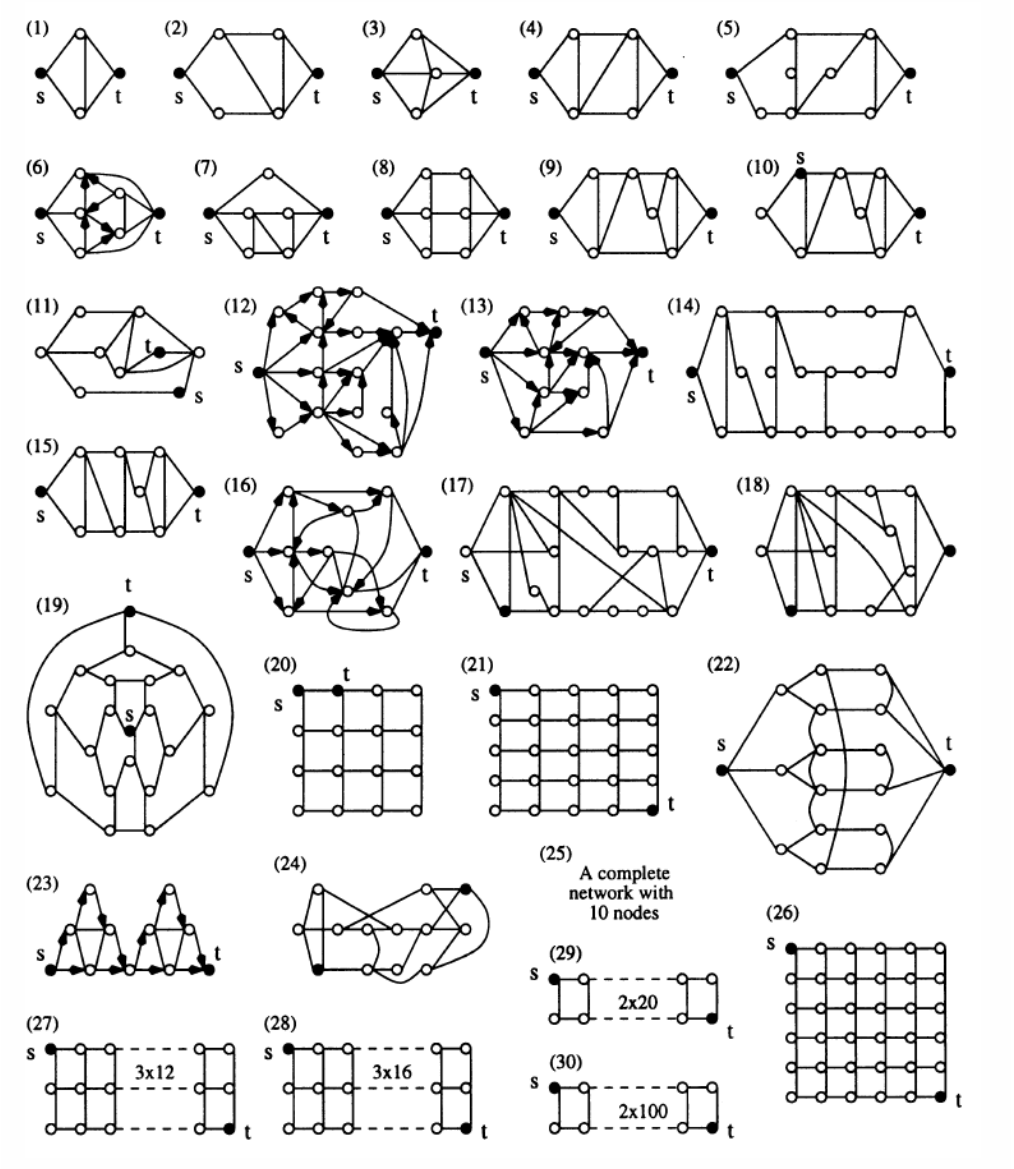
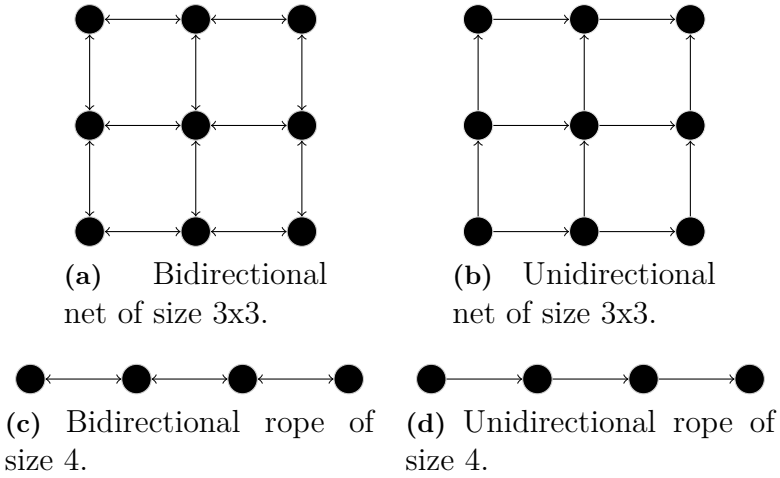


Figure 6.1: A collection of synthetic graphs created by Rauzy [26].

However, before executing the evaluation the type of all the edges are substituted to *north*. This because the matching code performs type checking (see Section 2.6) and therefore do not send messages to all adjacent vertices. This is prevented by having all edges of the same type. Thereby the partial matches will now be sent to all neighbours and the partial matches will increase with the number of edges as was the intent with increasing degree.

6.3.1 Generating Synthetic Graphs

Using the meta-model presented in Figure 6.3, the graphs were generated using Henshin declarative rules and operational units (see Section 2.3) using the local parallelized Henshin interpreter. Figure 6.6 to 6.17 illustrates the rules and units used for



(a) Bidirectional net of size 3x3.

(b) Unidirectional net of size 3x3.

(c) Bidirectional rope of size 4.

(d) Unidirectional rope of size 4.

Figure 6.2: The structure of the synthetic graphs generated. A regular grid structure ((a) and (a)) and a line structure ((c) and (d)). Both bidirectional or unidirectional.

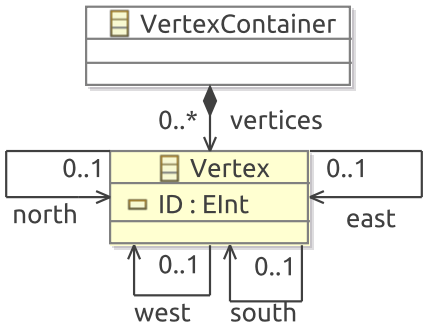


Figure 6.3: The meta model of the synthetic graphs.

the generation. The figures are accompaniment with in depth descriptions. However in order to create an overview the overall procedures are first presented.

Figure 6.4 illustrates the overall procedure for generating a three vertices large *rope* graph. Sub-figure 6.4a shows the initial source graph, Sub-figure 6.4b the result after extending the graph using the rule and lastly, Sub-figure 6.4c shows the result after making the edges bidirectional.

To generate the *net* graph involves more steps. Figure 6.4 illustrates the procedure of creating a 3 times 3 vertices large *net* graph. Sub-figure 6.5a shows the initial source graph, Sub-figure 6.5b illustrates the result of extending it *north*-direction. Sub-Figure 6.5c how it concurrently grows *east* and Sub-Figure 6.5d how it is filled. Sub-Figure 6.5e and 6.5f how it is made bidirectional by creating *west* and *south* edges.

Figure 6.6 illustrates the sequential unit *CreateRope*. The units has one argument which is the size n of the rope which should be created. The unit has one sub-rule

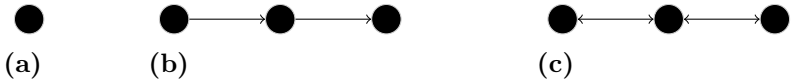


Figure 6.4: An illustration of the creation of a rope. (a) shows the initial source graph, (b)) the result after executing *NorthIterated(3)* and ((b) the optional step of making it bidirectional by executing *South*.

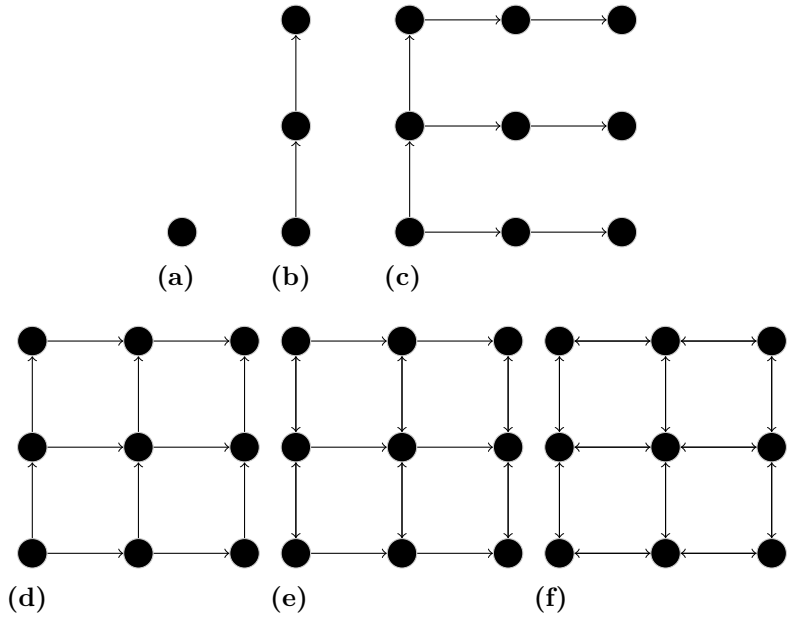


Figure 6.5: An illustration of the creation of a *net*. (a) shows the initial source graph, (b)) extending it *north*-direction. (c) how it concurrently grows *east* and (d) how it is filled. (e) and (f) shows how edges are made bidirectional.

which is supplied with the argument n when called.

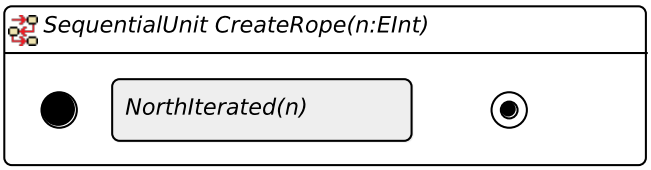


Figure 6.6: Henshin units to create a rope.

Figure 6.7 illustrates the iterative unit *NorthIterated*. The unit takes one argument n. The unit iterates n times; every time calling a rule called *North*. The annotation *[n-1]* refers to the mapping from the the argument of *CreateRope* to this rule. The argument should be one less because it is executed on a graph which already has one vertex.

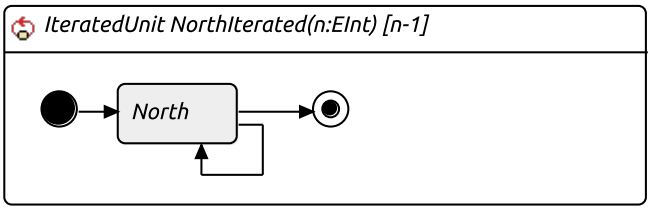


Figure 6.7: The Henshin iterated unit *NorthIterated*.

North is the last unit in the call stack for creating a rope and is illustrated by Figure 6.8. The rule has three vertices of which two are located in the LHS and three in the RHS. A match is found when a vertex of type *Vertex* has no edge of type north to another vertex of type Vertex. When a match is found a *north* edge is created to a new vertex of type *Vertex*. The annotation *@VertexContainer* is a method of specifying that all edges should be connected to the same *VertexContainer*. Therefore an edge of type *vertices* is created to all new nodes from the *VertexContainer* already connected to the existing vertices. If this rule is called multiple times a *rope*-like structure will emerge.

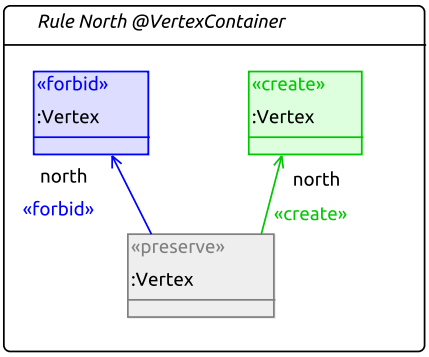


Figure 6.8: The Henshin rule *North*.

The result from the rule *CreateRope* (see Figure 6.6) is used in order to to create a bidirectional rope. Figure 6.9 illustrates the sequential rule *CreateBidirectionalRope* which does this. The rule takes an argument. Again, the length of the rope. First *CreateRope* is called supplied with the argument in order to create a rope. Then *South* is called in order to make it bidirectional.

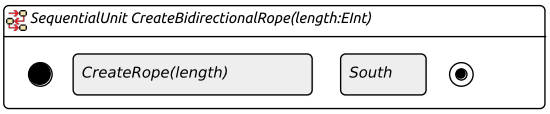


Figure 6.9: Henshin unit *CreateBidirectionalRope*.

Figure 6.10 illustrates the declarative rule *South*. The rule illustrates the concept of multirules as presented in Section 2.2. It consists of a single multirule containing two vertices and three edges. The pattern to search for, the LHS, consists of two vertices of type *Vertex*. These should be connected with an edge of type north (marked with «north*») but not connected in the other direction of an edge of type *south* (marked with «forbid*»). On all matches found a new edge of type *south* is created.

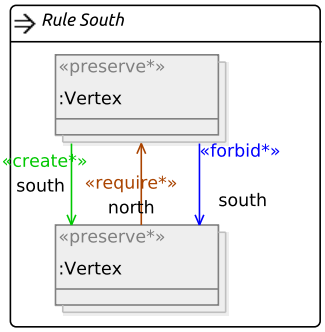


Figure 6.10: Henshin rule *South*.

The procedure of creating a net is more complex. Figure 6.11 illustrates the top-level Henshin sequential unit *CreateNet* used for generating the unidirectional net. It is a sequential unit which calls *NorthIterated* (see Figure 6.7), *EastIterated* and *FillIterated*. These are further presented below.

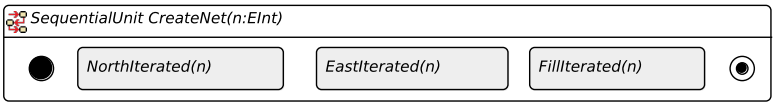


Figure 6.11: The Henshin unit used to generate the net graph.

When *NorthIterated* has been called the graph has the structure of a unidirectional rope. On this structure the rule *EastIterated* called. The rule is illustrated by Figure 6.12. *EastIterated* is an iterated unit which calls the rule *East* multiple times.

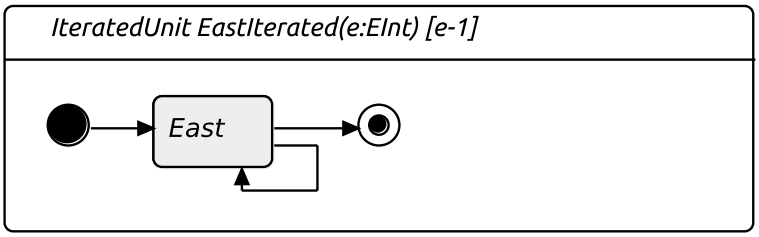


Figure 6.12: Henshin unit *EastIterated*.

The rule *East* is illustrated by Figure 6.13. The rule has the purpose of creating an *east* edge and connecting it to a new vertex for every vertex of type Vertex which do not already have such. This is achieved by a multirule containing the negative application condition marked with «forbid*» containing an edge and a vertex. The VertexContainer, which is not a part of the multirule, is here explicitly connected to the new vertex created. When executing this rule n times strings of size n will emerge in east-direction as was illustrated by Figure 6.5c.

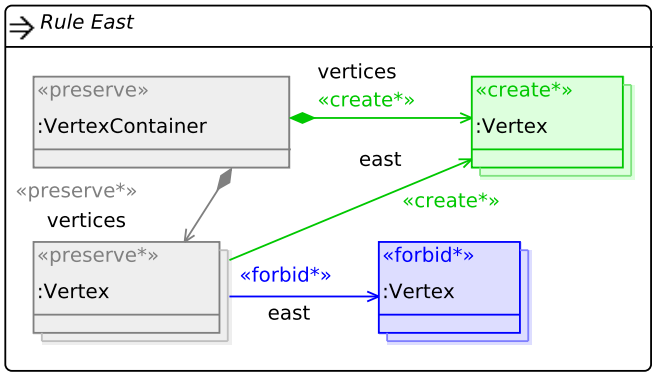


Figure 6.13: Henshin rule *East*.

In order to knit these strings and create a complete net, *FillIterated* is called. Figure 6.14 illustrates the iterated unit *FillIterated*. It calls *Fill* n times. Figure 6.5d illustrates this.

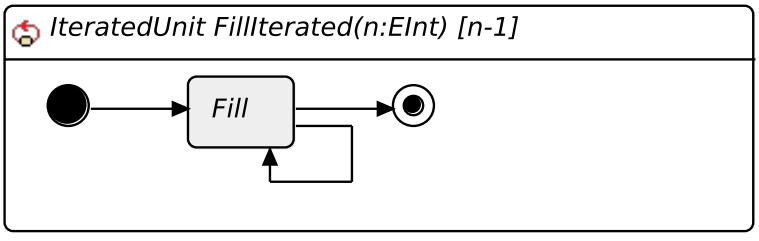


Figure 6.14: Henshin units *FillIterated*.

Fill is the final rule used to create the unidirectional net. It is illustrated by Figure 6.15. The rule consists of a single multirule which contains four vertices. These

vertices, of type Vertex, are connect like an open regularly formed quadrangle. The edges consist of two parallel east edges and an edge of type *north*. The purpose of the rule is to create the fourth edge in every such quadrangle (fill it). These is done via the NAC marked with «forbid*» and the edge to be create which is marked with «create*». When this rule is called several times inside the iterated unit it fills the net from west to east until the rule is no longer applicable.

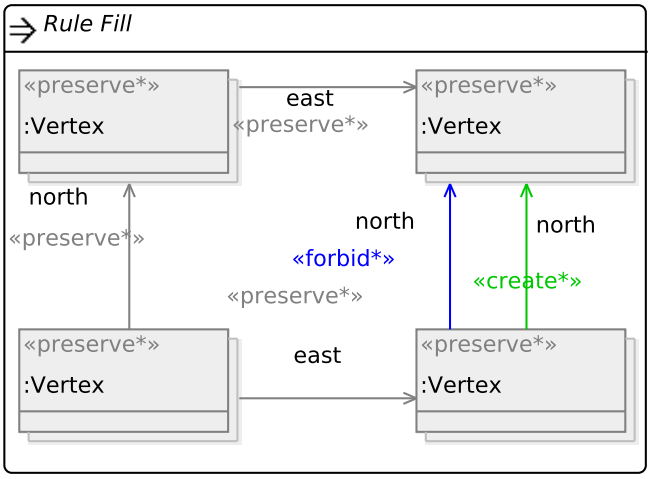


Figure 6.15: Henshin rule *Fill*.

The last structure, the bidirectional net, is created using the sequential unit *CreateBidirectionalNet* which is illustrated by Figure 6.16. The rule takes one argument which is the length of one of the sides of the net n. The net will be of the size n times n. First it calls CreateNet in order to create a unidirectional net. Then the rule *South* (see Figure 6.10) is called in order to create a reverse direction edge for every *north* edge. Lastly, *West* is called to in order to create the same for every edge of type *east*. Figure 6.17 illustrates the rule West.

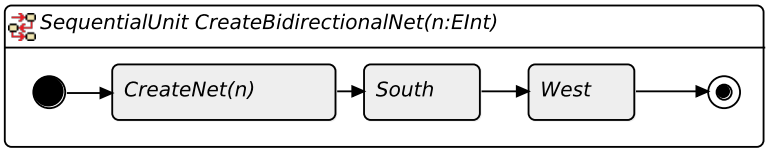


Figure 6.16: Henshin unit used to generate the bidirectional net graphs.

Even though the concept is simple the time required for creation of the graphs using the above rules is in the magnitude of days when generating structures consisting of millions of nodes. Therefore, it can be easily realized that graphs of this magnitude must be handled in a cluster instead of on a single machine. When the graphs were created they were converted into the required format of the code generated for the Henshin Giraph algorithm.

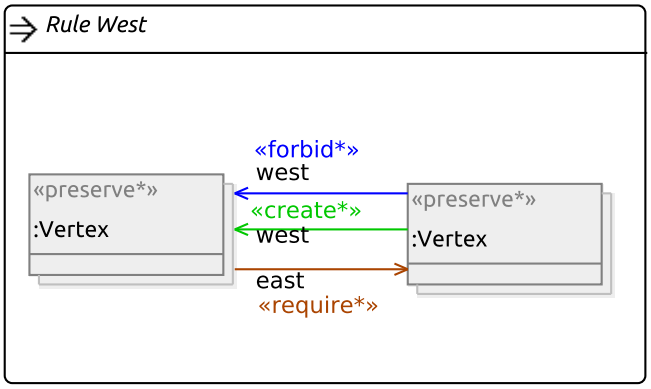


Figure 6.17: Henshin rule *West*.

6.4 Graph Transformation Rules

Two kinds of rules are used to execute on the in Section 6.3 presented source graphs. For the real data examples the previously described rule *AddCouples* is used. Figure 6.18 summarizes the rule and its sub-rules *CreateCouple* and *CreateOccurrences*. For the synthetic graphs has a rule been developed which is described below.

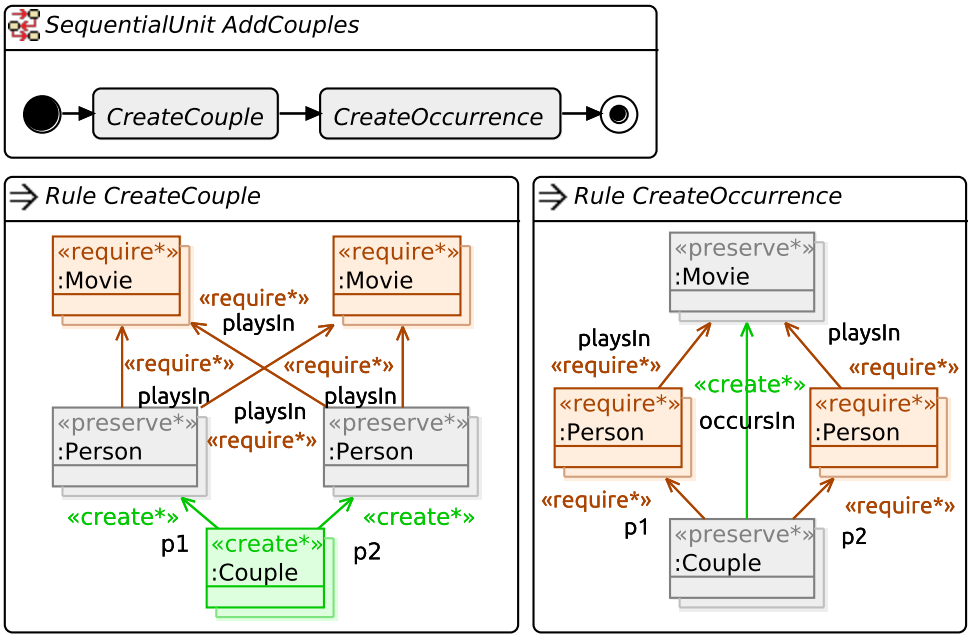


Figure 6.18: The Henshin rule *AddCouples*.

The rule to execute on the synthetic source graphs are developed to complement the *AddCouples* rule and show on additional weaknesses and strengths of the algorithm. The graphical appearance of the rules are presented in Figure 6.19. The rules are called *ForRopeX* where X determines the size of the number of vertices in its LHS. X in this case has been selected to be 1, 3 and 9.

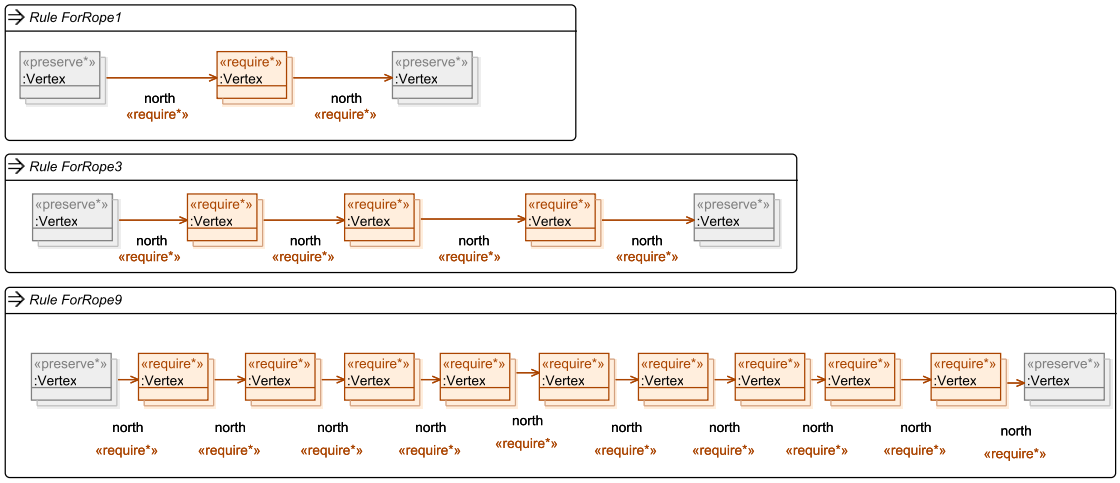


Figure 6.19: The Henshin rules *ForRope1*, *ForRope3* and *ForRope9*.

The reason for selecting such simple rules that they should have many matches and illustrate what happens when match sizes grows. Therefore, they have three different sizes. This for generating different amount and sizes of the partial matches and thereby have higher requirements of memory and communication bandwidth. When the rules are executed on the source graphs matches will be iteratively built up. Executing it on the rope graphs will have approximately the same number of partial matches seen over multiple microsteps. While executing it on the *net* will generate possible problems with increasing number of matches over the microsteps.

To further explain this concept look at a vertex of type *Vertex* in the middle of a net. The vertex will send messages to all adjacent vertices. The next steps will do the same. And then again the same, apart from removing injective matches. This will go on in the number of microsteps the rule has. The number of microsteps are the same as the number of vertices in the rule. This will lead to the number of partial matches growing exponentially and hence also the memory consumption.

If the partial matches removed due to injectivity checks are disregarded it is possible to derive a simple formula to describe the growth of the partial matches per vertex of the graph. The average degree of the graph is denoted d and the number of the current microstep in the *ForRopeX*-rule is denoted s. If the resulting number of partial matches is denoted m the formula will look like: $m = d^s$. The formula is not exact however suites as an upper bound. It for example do not describe what happens at the edges of the graph were vertices have a lower degree.

The microsteps in the *ForRopeX*-rules will be similar to each other and also have no rule to apply, only match. The real data example provides a full fledged example as presented in Section 2.6. This combination will enable edge case testing as well as normal execution.

To make the set of transformation rules to apply further more comprehensive a second type of rule were developed. The rule is the *ForNetX*-rule which is illustrated by Figure 6.20. The Figure gives Three sizes which are denoted *ForNet1*, *ForNet3*

and *ForNet9*. The number indicate how many vertices marked with *«require*»* resides in-between the vertices marked *«preserve*»*.

The main reason behind the creation of these rules is that they provide a high amount of partial matches. However, they should also have a small amount of final matches.

The rules take an amount of microsteps which is equal to the number of vertices in the transformation rule plus 1. In the first microstep the vertex in the lower left corner is checked and thereafter the execution iteratively matches all vertices on the left until it finds the vertex to preserve in the upper right corner. After this the partial matches are sent back to the vertex in the lower right corner to iteratively try to find the path on the right hand side of the figure. The sending back is the reason for the extra microstep. Finally the nodes in the PAC are removed and the set of the final matches is created.

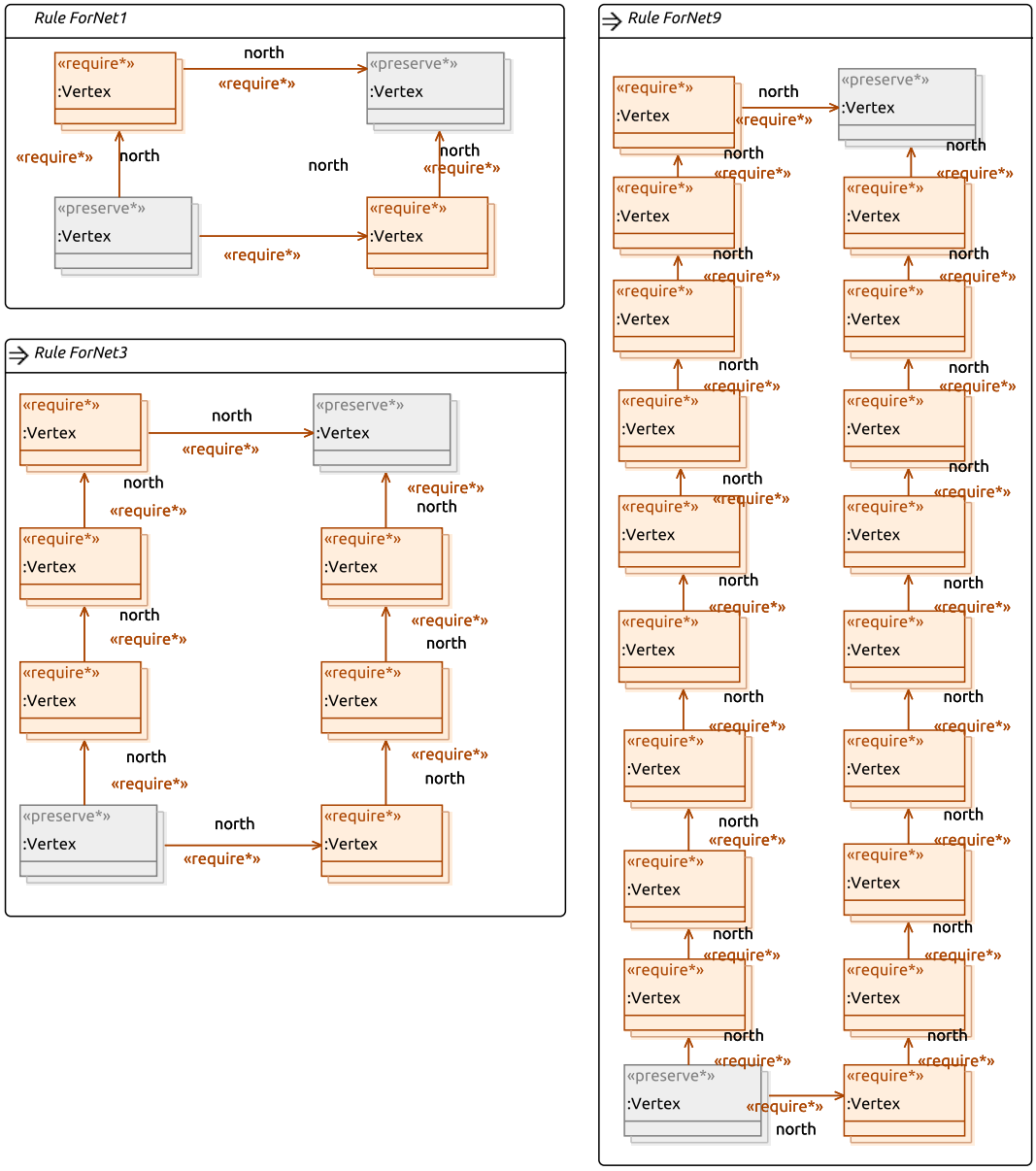


Figure 6.20: The Henshin rules *ForNet1*, *ForNet3* and *ForNet9*.

7

Implementation

This chapter aims to present and describe the matching algorithms generated using the new implementation of the code generator. The new implementation are compared with the original implementation (see Section 2.6) using a running example. The running example is a declarative Henshin rule which is is executed on a small target graph. The rule *CreateCouple* (Figure 2.3) is used and the detailed description of it provided in Section 2.6 is used as a base-point for the discussion. It is the first sub-rule of the *AddCouples* (see Figure 6.18) sequential operational unit which resides in the benchmarking suite (Chapter 6).

The solution was created in order to mitigate the problems with the same resource requirements for every vertex in each superstep of the BSP model as was discussed in Section 5.2. Recap Figure 2.7 which illustrates that every vertex executes the same microstep in the same superstep. Therefore, the new solution tries to break this symmetry by decoupling the microstep from the superstep and thereby spread out the resource requirements.

To manage this, the iterative matching procedure for the vertices is started in a number of different stages depending on the vertex's id. When the vertex has started calculating, the microstep will be tied to the partial match and no longer to the overall calculation. Figure 7.1 illustrates this concept when executed using the *CreateCouples* rule and the minimal source graph in Figure 2.6. The concept can be resembled with a staircase of size n. In the figure n is equal to three.

The main disadvantage of this solution is that this increases the number of supersteps by the number of extra stairs, which in the illustration (see Figure 7.1) is two. However, these steps are going to require considerably less time. Therefore, the spread-out of the resource requirements is assumed to result in on average less resources used over the supersteps and thereby speed up the overall algorithm in environments with sparse resources. This will then possibly lead to a net benefit.

In order to make this possible the data type used for storing the partial matches needed to be changed. In Section 2.6 the previous match structure was discussed. The previous data type had no information about in which microstep the match was created. A microstep simply assumed that the correct type of match was received. In order to be able to have multiple different microsteps executing at the same time, possibly for the same vertex, a match-specific microstep needed to be stored in each match. Therefore, the preamble of the match was extended to also store the

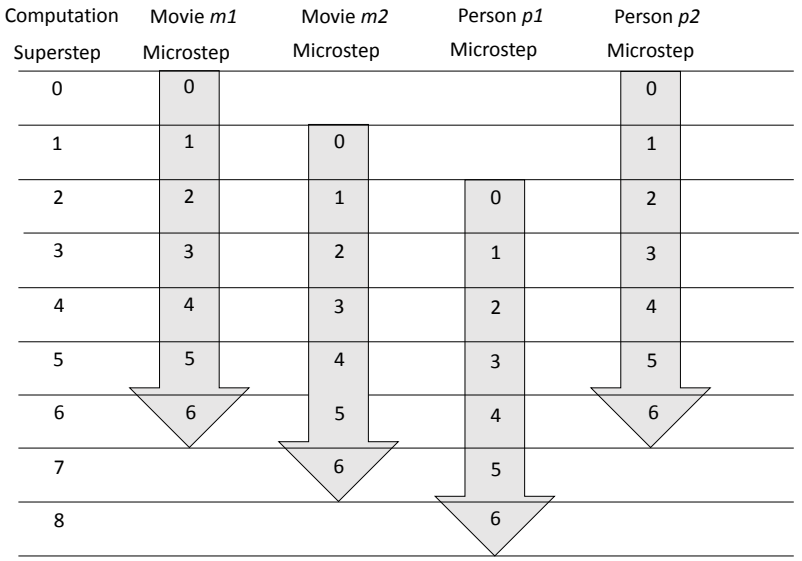


Figure 7.1: The figure illustrates the concept of a staircase in the start-up phase of the algorithm. In the picture three stairs exists.

microstep for the match.

The downside of storing the microstep in every match it result in 1 byte larger partial matches. The match already stores 4 bytes for storing the segments and the amount of bytes used for storing a variably sized integer for every vertex in the match. Therefore, this single byte can be neglected. The match will now conceptually look like this: *segmentNumber,microstep:[vertex1,vertex2,vertex3...]*.

In order to easily describe the execution behaviour of the alternative solution a simple graph containing the least amount of matches possible is used as a target graph for the execution of the transformation. The graph contains four vertices, two of each type, and four edges, each of the same type. It is illustrated by the Henshin rule in Figure 2.6 on page 18. The rule to execute is the *CreateCouple* in Figure 2.3 on page 2.3.

Table 7.1 illustrates the execution in the same manner as was done in Section 2.6. The difference here is the notation $x : [a, b...]$ where x here is the match-specific next microstep to be executed on the match. Three stairs was used when executing the example and as can be seen matches with two different microsteps exists in some supersteps, e.g. microstep 2 and microstep 4 in superstep 3. The reason for only two is that there are only two *Person*s in the source graph. What also can be noted from the table is that the rule is not applied until the last superstep in order to preserve the correctness of the graph transformation. If that was not done more or less matches should have been found for some rules.

As also can be noted from Table 7.1, superstep 7 is the only superstep performing a merge (join) of multiple matches. It was realized that the join with the current search-plan had to have all partial matches for that particular microstep available in order to correctly function. To realize this all vertices had to accumulate all the matches and wait for all stairs to reach microstep 5. This is the explanation for all

Superstep	Vertex	Create	Extend	Merge	Forward	Apply	Delete	Send to	Matches
0	p2	1:[p2]						m1,m2	1
1	m2		2:[p2,m2]					p2	
	m1		2:[p2,m1]					p2	2
2	p1	1:[p1]						m1,m2	
	p2				3:[p2,m2]			m1,m2	
	p2				3:[p2,m1]			m1,m2	3
3	m1		2:[p1,m1]					p1	
	m2		2:[p1,m2]					p1	
	p1	4:[]						p1	
	p2	4:[]						p2	
	m1	4:[]						m1	
	m2	4:[]						m2	
	m1						4:[p2,m2,m1]		
	m2						4:[p2,m2,m2]		
	m1						4:[p2,m1,m1]		
	m2		4:[p2,m1,m2]					m1	3
4	p1		5:[p1]					m1,m2	
	p1				3:[p1,m2]			m1,m2	
	p1				3:[p1,m1]			m1,m2	
	m1						5:[]		
	m2						5:[]		
	p2		5:[p2]					m1,m2	
	m1				5:[p2,m1,m2]			m1	5
5	m1				5:[p1]			m1	
	m2				5:[p1]			m2	
	m1				5:[p2]			m1	
	m2				5:[p2]			m2	
	m1				5:[p2,m1,m2]			m1	
	m1						4:[p1,m2,m1]		
	m2						4:[p1,m2,m2]		
	m1						4:[p1,m1,m1]		
	m2		4:[p1,m1,m2]					m1	6
6	m1				5:[p1,m1,m2]			m1	
	m1				5:[p1]			m1	
	m2				5:[p1]			m2	
	m1				5:[p2]			m1	
	m2				5:[p2]			m2	
	m1				5:[p2,m1,m2]			m1	6
7	m1			6:[p1,m1,m2,p2]				p2	
	m1			6:[p2,m1,m2,p1]				p1	
	m2						5:[p1]		
	m2						5:[p2]		2
8	p1					7:[p2,p1]			
	p1					7:[p1,p2]			2

Table 7.1: An execution example of *CreateCouple* using the alternative implementation.

matches being of microstep 5 in superstep 6 as can be seen in the table.

The accumulation of matches due to the *join*-step and the delaying of the rule application until the last superstep results in in-total more partial matches than the previous solution. However, these partial matches are sent from the sending vertex to itself. This is most likely less expensive than sending to another node and might

therefore not have such a large impact.

The code for the matching algorithm has some major differences. Firstly, in order to provide the stepwise start-up the initial *if*-clause checks whether this vertex should start or not. Secondly, the microstep is now match-specific instead of being equal to the overall superstep which is achieved by storing it in the match. Lastly, in order to join different branches in the left hand side with directed edges (see Figure 2.3) the whole algorithm needs to be synchronized by waiting for all steps to arrive before joining.

The first microstep of the code generated using *CreateCouple* is the only step which is dependent on the overall microstep in combination with the id of the vertex. Listing 7.1 lists the code for microstep 0 (compare Listing 2.6). The code is surrounded with an *if*-clause. The purpose of it is to start approximately $1/n$ of the total amount of vertices at each superstep. n is stored in the constant *STAIR_COUNT*. The method *getPart* on the first line divides the vertices in n disjoint steps. Lines 6 and 7 are also changed where now the microstep is stored in the match. After leaving a microstep it should have the number of the next microstep in the match which is assured by line 7. In Table 7.1 this microstep is executed for *p2* in superstep 0 and for *p1* in superstep 2.

Listing 7.1: Microstep 0.

```
1  if (microstep < STAIR_COUNT && getPart(vertex.getId()) == microstep){
2      ok = vertex.getValue().get() == TYPE_PERSON || vertex.getValue().get() ==
           TYPE_ACTOR || vertex.getValue().get() == TYPE_ACTRESS;
3      ok = ok && vertex.getNumEdges() >= 2;
4      ok = ok && (SEGMENT_COUNT == 1 || getSegment(vertex.getId()) == segment);
5      if (ok) {
6          Match match = new Match(segment, 0).append(vertex.getId());
7          match = match.setMicrostep((byte) (match.getMicrostep() + 1));
8          targets = new HashSet<>();
9          for (Edge<VertexId, ByteWritable> edge : vertex.getEdges())
10             if (edge.getValue().get() == TYPE_PERSON_PLAYS_IN && targets.add(edge.
                   getTargetVertexId()))
11                 sendMessage(edge.getTargetVertexId(), match);
12   }
13 }
```

Listing 7.2 shows the overall structure of the iterative matching procedure for the rest of the microsteps. Comparing to Listing 2.5 the structure now consists of a loop on the top level iterating over all the matches and then checking the microstep of the current match to perform the correct actions. The following listings will go through the rest of the iterative matching procedure and point out differences.

Listing 7.2: Overall matching structure.

```
Iterator<Match> matchesIterator = matches.iterator();
Match match = null;
while(matchesIterator.hasNext()){
  match = matchesIterator.next();
    switch ((int)match.getMicrostep()){
  case 1: ...
     break;
  case 2: ...
     break;
  case 3: ...
     break;
  case 4: ...
```

```
      break;
    case 5: ...
      break;
    case 6: ...
      break;
    default: ...
    }
}
```

Listing 7.3 provides the code for microstep 1. The difference compared to the old implementation is only that iteration is not performed over the matches and line 6 which increases the microstep stored in the match. In Table 7.1 this step is executed in superstep 1 and 3.

Listing 7.3: Microstep 1.

```
1 ok = vertex.getValue().get() == TYPE_MOVIE;
2 if (ok) {
3   match = match.append(vertex.getId());
4   if (!match.isInjective())
5     break;
6   match = match.setMicrostep((byte) (match.getMicrostep() + 1));
7   sendMessage(match.getVertexId(0), match);
8 }
```

The next microstep, number 2, is not particularly different either. The microstep is shown in Listing 7.4. The only difference, apart from not iterating over the matches, is that it increases the microstep stored in the match. In Table 7.1 the microstep is executed when for example *p1* forwards a partial match to *m1* and to *m2* in superstep 4.

Listing 7.4: Microstep 2.

```
1 match = match.setMicrostep((byte) (match.getMicrostep() + 1));
2 targets = new HashSet<>();
3 for (Edge<VertexId, ByteWritable> edge : vertex.getEdges())
4     if (edge.getValue().get() == TYPE_PERSON_PLAYS_IN && targets.add(edge.
          getTargetVertexId()))
5       sendMessage(edge.getTargetVertexId(), match);
```

Listing 7.5 presents microstep 3 which has some differences to the original implementation. However, as the next microstep creates a new match and not simply extends the incoming matches this step also has a separate *if*-clause which sends an empty message to be picked up in the next microstep. This because now the microstep is not executed if it is not needed. Listing 7.6 lists this extra step. Microstep 3 is also different by not iterating over the matches and by increasing the microstep stored in the match (line 8). In Table 7.1 this step is executed in order to create empty matches in superstep 3. In this superstep matches are also extended and deleted in the execution of the microstep.

Listing 7.5: Microstep 3.

```
1 ok = vertex.getValue().get() == TYPE_MOVIE;
2 if (ok) {
3   match = match.append(vertex.getId());
4   if (!match.isInjective())
5     break;
6   if (vertex.getId().compareTo(match.getVertexId(1)) < 0)
7     break;
8   match = match.setMicrostep((byte) (match.getMicrostep() + 1));
```

```
9   sendMessage(match.getVertexId(1), match);
10 }
```

Listing 7.6: Microstep 1.

```
1     if (microstep == 3)
2         sendMessage(vertex.getId(), new Match(segment, 4));
```

Microstep 4, which is presented in Listing 7.7, is the reason for the extra step which sends an empty match. However, some vertices receive real partial matches from previous microstep. These are of type *Person*. Lines 1-5 makes the vertex send them to itself to be handled in the next microstep. However, the second part of the code should check for *Person*s fulfilling the conditions of the second *Person* in the rule. This is done as in the original implementation (see Listing 2.11), however three lines are added and one changed. In order to only execute this for the empty match and not multiple times if also other matches are received line 9-10 checks and breaks if the match is not empty. Line 11 and 12 are changed and new in order to incorporate the microstep number in the match. In Table 7.1, two new matches is created and to one match is forwarded in superstep 4 while executing this microstep.

Listing 7.7: Microstep 4.

```
1  id = match.getVertexId(1);
2  if (vertex.getId().equals(id)) {
3    match = match.setMicrostep((byte) (match.getMicrostep() + 1));
4    sendMessage(id, match);
5  }
6    ok = vertex.getValue().get() == TYPE_PERSON ||vertex.getValue().get() ==
         TYPE_ACTOR || vertex.getValue().get() == TYPE_ACTRESS;
7    ok = ok && vertex.getNumEdges() >= 2;
8    if (ok) {
9      if (!(match.getMatchSize() == 0))
10       break;
11     match = new Match(segment, 4).append(vertex.getId());
12     match = match.setMicrostep((byte) (match.getMicrostep() + 1));
13     targets = new HashSet<>();
14     for (Edge<VertexId, ByteWritable> edge : vertex.getEdges())
15       if (edge.getValue().get() == TYPE_PERSON_PLAYS_IN && targets.add(edge.
             getTargetVertexId()))
16         sendMessage(edge.getTargetVertexId(), match);
17       }
```

Listing 7.8 lists microstep 5. Recall that this step is a *join* and that it takes the most time as presented in Section 5.2. In order to properly join all the matches the microstep needs to wait for all matches reaching this microstep. This is done in lines 1-4 which checks if the partial matches created in the last stair has arrived here yet. If they have not the match is sent to itself and the microstep is halted to be taken up next superstep. When all matches are of microstep 5 the rest of the microstep is executed. Apart from the storage of the increase of the microstep number the rest of the step functions the same as the original implementation.

This step is the reason for multiple matches of the same microstep across supersteps Table 7.1. In superstep 5 the microstep is executed for the first time. Here four matches are accumulated. The next step also send the matches of microstep 5 to itself. In superstep 7 the joining is finally performed which creates two matches.

Listing 7.8: Microstep 5.

```
1  if (microstep < 5 + STAIR_COUNT - 1){
2    sendMessage(vertex.getId(), match);
3    break;
4  }
5  id = vertex.getId();
6  if (id.equals(match.getVertexId(1)))
7    matches1.add(match.copy());
8  else
9    matches2.add(match.copy());
10 if (matchesIterator.hasNext())
11   break;
12 for (Match m1 : matches1)
13   for (Match m2 : matches2) {
14     match = m1.append(m2);
15     if (!match.isInjective()) {
16       continue;
17     }
18     match = match.setMicrostep((byte) (match.getMicrostep() + 1));
19     sendMessage(match.getVertexId(3), match);
20   }
```

The last microstep, number 6, is presented in Listing 7.9. As previously states all vertices needs to wait until the matching procedure is finished for all stairs. Therefore, line 7 has an extra check which ensures this. However, in this specific case this check is not needed since the previous step was a *join*-step which required synchronization. In Table 7.1, this can be seen in superstep 8 which removes the PACs from the match and applies the rule on the final set of matches.

Listing 7.9: Microstep 6.

```
1  targetId = match.getVertexId(2);
2  for (Edge<VertexId, ByteWritable> edge : vertex.getEdges())
3    if (edge.getValue().get() == TYPE_PERSON_PLAYS_IN && edge.getTargetVertexId().
         equals(targetId)) {
4      match = match.remove(2).remove(1);
5      if (finalMatches.add(match)) {
6        match = match.setMicrostep((byte) (match.getMicrostep() + 1));
7        if (segment == SEGMENT_COUNT - 1 && microstep == 6 + STAIR_COUNT - 1 ) {
8          applyCreateCouple(vertex, match, appCount++);
9        } else {
10         sendMessage(vertex.getId(), match);
11       }
12     }
13     break;
14   }
```

Listing 7.10 lists the code in the default block of Listing 7.2. This code also has the function to make sure that the rule is applied only when all vertices are in the last microstep. However, it also has the function to filter out duplicate matches from the set of final matches. This is done via using the same hash set as in the last microstep. Line 1 does this. Again, this is not needed in this particular case because of the join and should have been optimized in the rewritten code generator.

Listing 7.10: Code executed while waiting for all microsteps to finish the matching procedure.

```
1        if (finalMatches.add(match))
2          match = match.setMicrostep((byte) (match.getMicrostep() + 1));
3          if (segment == SEGMENT_COUNT - 1 && microstep == 6 + STAIR_COUNT - 1 )
4            applyCreateCouple(vertex, match, appCount++);
5          else
6            sendMessage(vertex.getId(), match);
```

The static and dynamic semantics of the rules and units in the implemented alternative are equivalent to the non-parallel (non-BSP) local and the original version of the Henshin transformation interpreters. This is ensured using the currently available test suite consisting of 18 non-trivial transformation rules as well as comparing the output from the transformations in the benchmarking suite (Chapter 6).

Recall that the idea behind the solution presented was to even out the resource requirements across the supersteps by not executing the same microstep for every vertex of the graph. However, this did not fully work for the particular example of *CreateCouple* (see Figure 2.3). The reason for this is the need for a join of partial matches which was created by two from each other unreachable vertices. This created a need for synchronizing the execution of all the different *stairs*. This forced all the *stairs* to again execute the same microstep at the same time which possibly causes the problems which were discussed in Section 5.2. It also causes some extra complications in the form of extra partial matches.

However, in the supersteps before the synchronization is performed it is reasonable that the resource requirements are spread out across the supersteps. Also if an iterative matching procedure of a rule do not contain a *join*-step the result may be spread out across the superstep and thereby cause an overall performance increase. See for example the rules in Figure 6.19 on page 6.19 and Figure 6.20 on page 6.20 which do not map to any *join*s.

Also it is reasonable that the net benefit of applying this approach is dependent on how many microsteps it takes to apply the rule. If the rule has too few microsteps the fixed costs of using this approach is possibly not payed off. If also the number of partial matches is exponentially increasing, as was pointed out in Section 6.4, it is possibly that the net benefit is even higher.

8

Evaluation

Chapter 6 presented the benchmarking suite. This chapter aims to present and discuss the results from using it. Executable code for Giraph was generated using the original implementation of the code generator, as presented in Section 2.6, and the new implementation of the code generator, as described in Chapter 7. The evaluation thereby aims to compare the two groups by: Firstly, in Section 8.1 the results from evaluating using a real graph and rule example are presented. This in order to relate to the results by Krause et al [1]. Secondly, in Section 8.2 more comprehensive results from evaluating using the synthetic graphs and rules are presented. The number of *stairs* used for the new implementation is 3 throughout the evaluation.

8.1 Real Data

Executable code was generated for the Henshin unit *AddCouples* (see Figure 6.18 on page 51) utilizing the two implementations of the code generator. The code was executed on the cluster with the configuration presented in Section 6.4. The real data example presented in Section 6.3, the IMDb movies database, was used as the source graph. The number of compute nodes in the cluster was varied. Three executions per number of nodes were performed for each of the two groups. In total 60 executions were performed. Metric 1 (see Figure 6.1 on page 6.1) and Metric 2 (see Figure 6.2 on page 6.2) were then used on the complete sample. Section 8.1.1 presents and analyses the results from Metric 1 and Section 8.1.2 presents and analyses the results from Metric 2.

Interesting is that the author did not manage to get any of the two algorithms working on the Glenn cluster for less than 14 nodes. Less was tried, however the job was either killed by generous time limit or due to memory issues. This is an interesting contrast in comparison to the evaluation by Krause et al. [1] who did execute the same unit, on the same data, on the same cluster and using less nodes than 14. Possibly this is an effect of changed software stack on the cluster, however this cannot be verified. Therefore, the number of nodes used are 14-32, skipping the odd numbers.

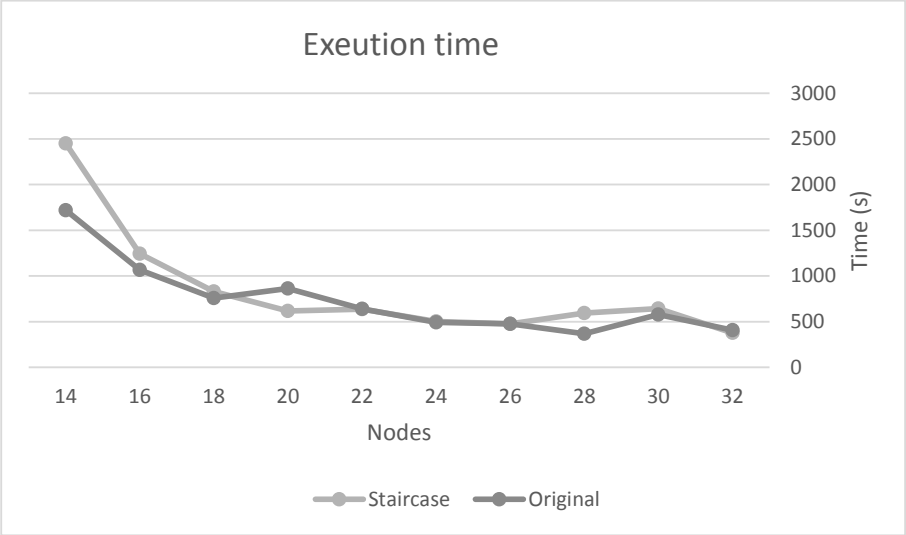


Figure 8.1: Metric 1 utilized on the execution of the two groups, *Staircase* and *Original*, of the unit *AddCouples* and varying the number of cluster nodes.

8.1.1 Metric 1

Figure 8.1 illustrates the results from the first metric. The mean value of the execution time for the three executions are presented for each number of cluster nodes. It is hard to see any apparent differences in total. However, the new implementation, marked *Staircase*, seems to perform worse for the lowest amount of nodes.

The mean values and standard deviations for the groups were $\mu_S = 836$ s and $s_S = 618$ s for the staircase implementation and $\mu_O = 737$ s and $s_O = 405$ ms for the original implementation. The overall mean difference and the sample standard deviation of the difference were $\mu_D = 99$ s respective $s_D = 323$ s.

Since the data is paired and a normal distribution cannot be assumed, a non-parametric paired Wilcoxon signed-rank test [27] was calculated in order to find out if there is a significant difference in their median values. The effect size was calculated using the Vargha and Delaney's $\hat{A}_{12}$ method [28]. The statistical tests were performed using R [29] and the package *effsize* [30] for calculating the effect size. The test was two-tailed and the null and alternative hypothesis are listed immediately below.

$$H_0 : m_d = 0$$
$$H_1 : m_d \neq 0$$

The results were $p = 0.1706$ and $\hat{A}_{12} = 0.47$. Using an α-level of 0.05 there is no significant difference. Therefore, the null hypothesis cannot be rejected. We cannot see a significant difference in computation time between the two groups.

The reason for no improvements in the performance in this case is likely to be the shortcoming of the current solution. As stated in Chapter 7 the solution has some problems when the search-plan consists of *join*-steps. When *join*-steps exists there is a need for synchronizing all the stairs of the algorithm. This means that the

resource requirements cannot be fully distributed over the supersteps. But this real data example constitutes a special case and this is not likely to be seen in many other examples.

8.1.2 Metric 2

Figure 8.2 illustrates the results from the second metric (see Figure 6.2 on page 6.2). The metric measures the total used node time, i.e. the execution time for the computation multiplied by the number of nodes utilized. Utilizing the metric, it is hard to see any apparent differences. It may be possible that the original version uses the resources more efficiently for a high number (30-32) of nodes which may explain why they perform rather equal when utilizing Metric 1 (see Figure 8.1). However, the sample size is to small to perform detailed testing for a particular number of nodes. Tests are instead performed for the whole sample.

The mean values and sample standard deviations for the groups were $\mu_S = 16776$ s and $s_S = 7448$ s for the staircase implementation and $\mu_O = 15101$ s and $s_O = 4459$ s for the original implementation. The mean difference and the standard deviation of the difference were $\mu_D = 1676$ s and $s_D = 6104$ ms.

Wilcoxon signed-rank test test was calculated for the sample containing all executions with the two groups. Vargha and Delaney's method for effect size was also calculated. The null and alternative hypothesis are:

$$H_0 : m_d = 0$$
$$H_1 : m_d \neq 0$$

The statistical testing resulted in $p = 0.1579$ and the effect size $\hat{A}_{12} = 0.46$. The effect was thereby negligible and the null hypothesis cannot be rejected at a $\alpha = 0.05$ level of significance. It cannot be proved that there is significant difference in their mean values.

8.2 Synthetic Graphs

Evaluation was also performed by using synthetic source graphs and transformation rules. Executable code for the synthetic transformation rules (see Section 6.4) was generated utilizing the two alternative implementations of the code generator. The transformation rules used are the *ForRopeX*-rules (see Figure 6.19 on page 52) and the *ForNetX*-rules (see Figure 6.20 on page 54). Each rule has three different sizes.

The source graphs used are the *rope*- and *net*-graphs presented in Section 6.3. Both the graph-types have bidirectional and unidirectional alternatives. The graphs were generated using the procedure described in Section 6.3.1. This resulted in *net*- and *rope*-graphs in the sizes 1.44, 1.69, 1.96 and 2.25 million vertices.

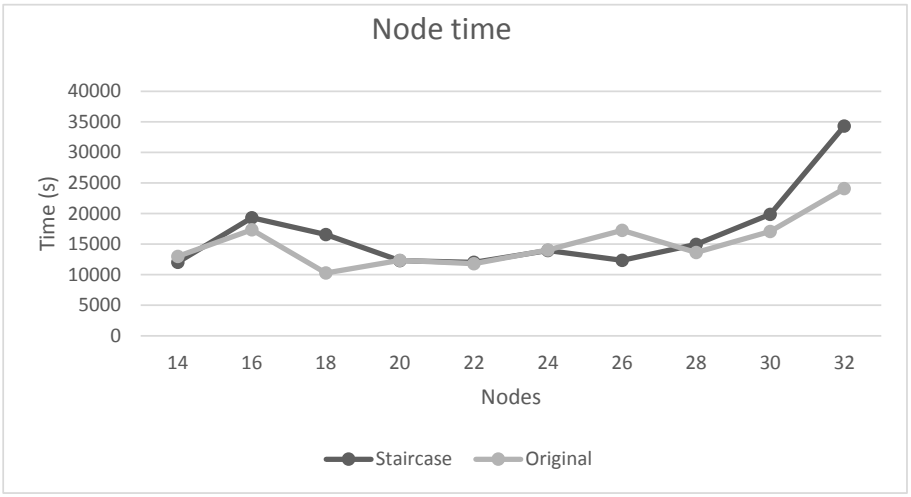


Figure 8.2: Execution of *AddCouples* using the two implementations taking Metric 2.

Two types of evaluation were performed. The first is presented in Section 8.2.1 and aims to meassure the execution time for different combinations of graphs and rules. The second is presented in Section 8.2.2 and aims to meassure and compare the memory usage of the two implementations by utilizing Metric 3 (see Figure 6.3 on page 42).

8.2.1 Execution Time

The initial idea was to execute all the different alternatives of the rules and graphs. However, because of the limited availability of the cluster that could not be achieved. The cluster usage is limited to 5000 core-hours a month. Since the machines have 16 CPU cores; one hour of execution for one single node equals 16 core-hours. Therefore, a subset of the rules and graphs had to be chosen in order to still preserve the key idea behind the evaluation as was presented in Chapter 6 and attain a sufficiently large sample in order to prove statistical difference.

Most importantly was the variation of the out-degree of the vertices and the size of the rules. Therefore, the size, counted in vertices, of the source graphs was set constant to 2.25 million vertices and only the *net*-graphs were used as source graphs. The degree was varied by using the same data-set in either the bidirectional or the unidirectional version. This resulted in an on average out-degree of 2 or 4 for the vertices in the source graphs. The number of rules were also minimized. Only the *ForRopeX*-rules (see Figure 6.19) were executed. However, this enabled variation of the size of the rules in three different sizes.

With the purpose of finding on optimal trade-off between being able to execute a sufficient number of trial executions and using as many nodes in the cluster as possible test trials were executed. During these trial executions it was impossible to get a result from the *ForRope9*-rule using up to 32 cluster nodes with the bidirectional

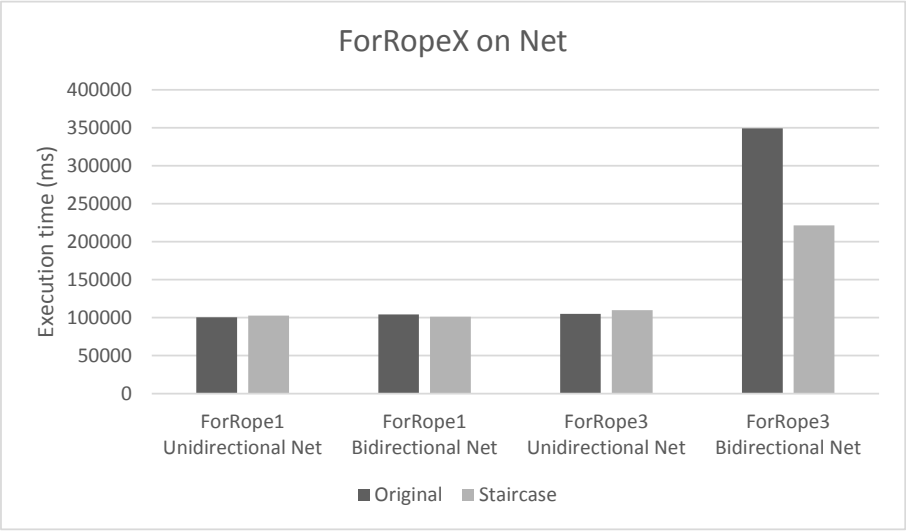


Figure 8.3: Results from execution of *ForRope1* and *ForRope3* on 2.25 million vertices large *nets* using 8 cluster nodes

net as the source graph for any of the implementations. The computation failed due to memory issues. Utilizing 16 cluster nodes and the unidirectional net, resulted in an execution time of 90 minutes for the *staircase*-solution and 170 minutes for the original solution which did not suit for multiple executions. This because one execution, which takes approximately 3 hours and utilizes 16 nodes results in a core-hours-consumption of 768 core-hours. That is simply too high. However, some executions were run using the spare resources after the rest of the evaluation was performed. These are reported on last in this section.

In order to be able to execute all rules together with all the graphs; only the *ForRope1* and *ForRope3* were used for the experiment. 8 nodes were selected as it was the lowest amount of nodes which could calculate a solution for all the combinations and utilized moderate time. 30 trial execution were performed for each of the 4 combinations for both groups. The resulting mean values are illustrated for comparison purpose in Figure 8.3.

From the figure it is apparent that when executing the large rule on the bidirectional source graph the *staircase*-alternative utilizes less time. The trial executions of the even larger graph transformation rule using 16 nodes further indicate that the difference might be even larger when larger transformation rules are executed. The reason for this might be the exponential growth of partial matches as was discussed in Section 6.4 and that the new solution succeeds in evening out the resource usage over the supersteps.

Figure 8.4 presents a box-plot were the execution time for the two groups are compared. The interquartile range is considerably smaller for the two groups while the median is approximately the same. The overall mean values and standard deviation of all 120 trials from each group were:

$$\mu_{\mathrm{O}} = 165 \text{ s and } s_{\mathrm{O}} = 139 \text{ s}$$

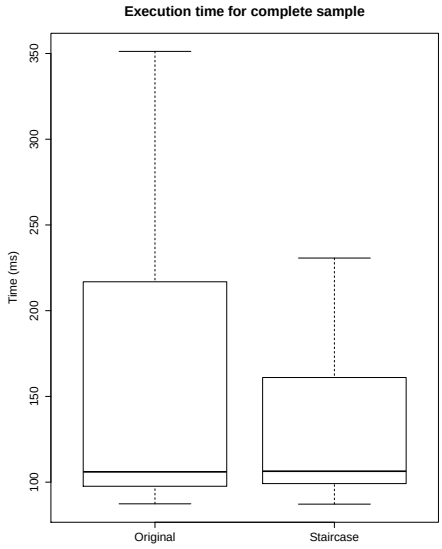


Figure 8.4: A box-plot of the sample containing the 240 executions on the synthetic graphs.

$$\mu_S = 134 \text{ s and } s_S = 56 \text{ s}$$

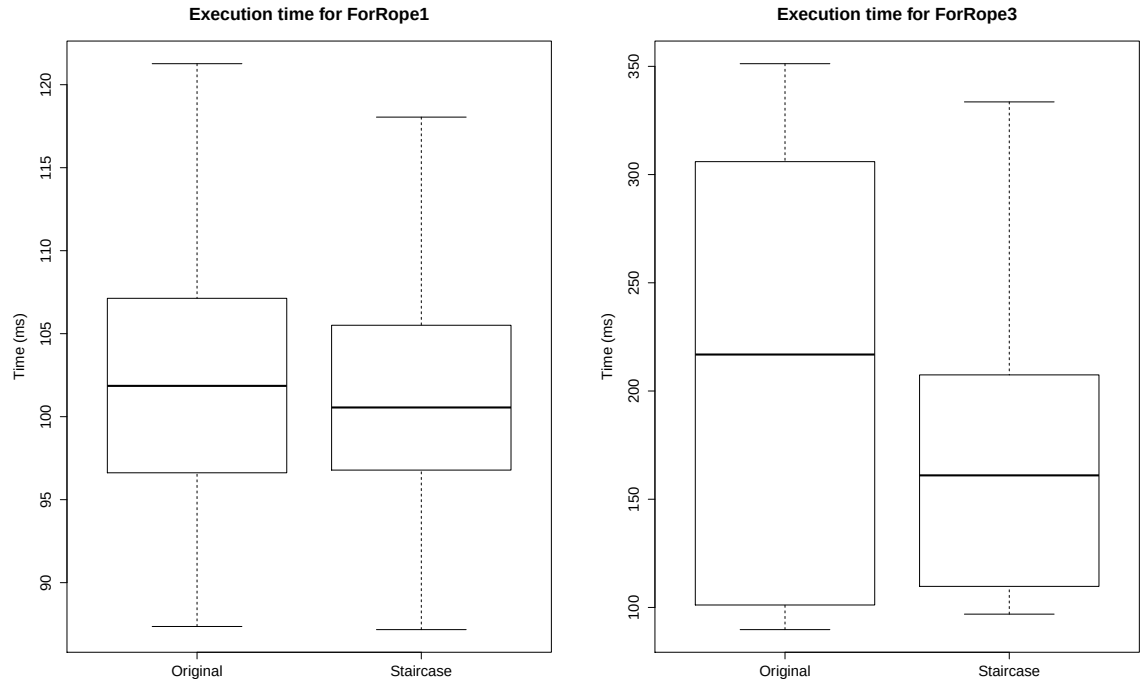
A one-tailed paired Wilcoxon signed-rank test was calculated for the sample containing all executions with the two groups. The reason for using the non-parametric test is that a normal distribution cannot be assumed and the reason for using the paired version is that the data are paired. Vargha and Delaney's method for effect size was also calculated. Each of the tests were performed with the below null and alternative hypothesis.

$$H_0 : m_O \leq m_S$$
$$H_1 : m_O > m_S$$

The complete set of data was tested. It resulted in a level of significance of $p = 0.002273$ and an effect size of $\hat{A}_{12} = 0.50$. The null hypothesis can thereby be rejected. The old solution in fact has a significant longer execution time than the new at an $\alpha = 0.05$ level of significance.

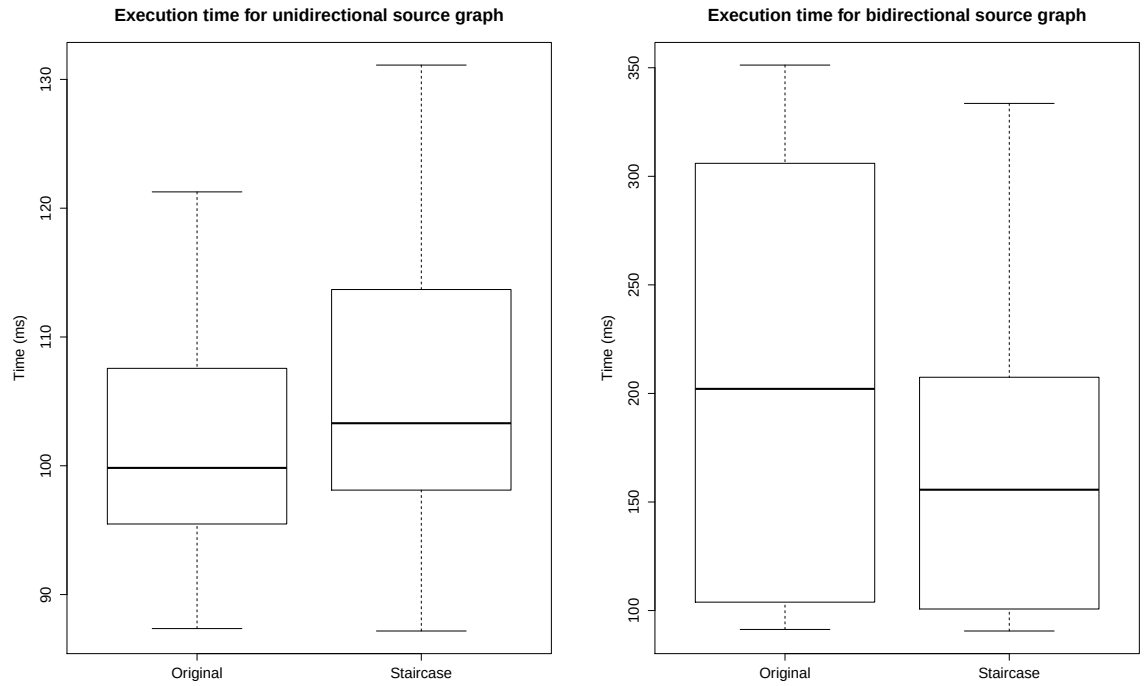
Figure 8.4 presents box-plots for the four combinations. Figure 8.5a and Figure 8.5b presents box-plots for *ForRope1* respective *ForRope3* and in Figure 8.5c and Figure 8.5d are box-plots for the unidirectional respective bidirectional nets presented. The interquartile ranges are smaller for the new implementation except for the unidirectional net. This also yields for the midhinge.

The statistical tests were also performed with the different rule sizes and different source graph sizes as constants. This because it was likely that graphs with different out-degree and rules with different number of vertices would perform differently. The new solution was also designed to handle large rules and source graphs with high out-degree more efficiently. The results are presented in Table 8.1. The results indicate a significant improvement for the new implementation, at $\alpha = 0.05$ level



(a)

(b)



(c)

(d)

Figure 8.5: Box-plots for the four different combinations of synthetic graphs and rules.

Constant	*ForRope1*	*ForRope3*	Unidirectional *net*	Bidirectional *net*
p-value	0.3884	9.078e-05	0.9715	8.706e-076
effect size	0.51	0.56	0.38	0.63

Table 8.1: Effect size and p-value calculated for different rule and graph sizes.

of significance, for the bidirectional net and the larger rule. However, there is no significant improvement for the small rule and the unidirectional net. The effect sizes are small.

The results are interesting though they clearly shows the effect of what happens when the number of partial matches grows. Both for the larger rule and the source graph with the higher out-degree the number of partial matches are considerably higher and grows considerably faster between the microsteps in the iterative matching procedure. It appears that the new solution has succeeded in even out the number of partial matches over the supersteps and thereby managed to decrease the resource consumption. Either memory consumption or bandwidth usage has been decreased. For further discussion see the previously presented qualitative analyses in Chapter 5.

On the other hand, the new implementation has increased fixed costs in terms of larger partial matches and longer execution counted in the number of supersteps. It might be possible that this do not pay off for either small rules are for sparse source graphs which are the cases in rule *ForRope1* and the unidirectional net.

Since the trend seems to be very clear that transformations rules with a larger number of nodes have a higher performance when executed using the new algorithm six new computations were made. This time using the rule *ForRope9*, 16 cluster nodes and executing 3 times per implementation. The mean values were $\mu_{\mathrm{O}} = 9619$ s and $\mu_{\mathrm{S}} = 5419$ and the standard deviations $s = 705$ s and $s = 381$ s. A one-tailed Wilcoxon signed rank test reported a level of significance of $p = 0.05$. The effect size was $\hat{\mathrm{A}}_{12} = 1$ which is large. The new implementation in fact perform even better for large transformation rules.

8.2.2 Memory Usage

In Chapter 7 it was discussed that the new solution might require less memory. A memory test was therefore performed in order to validate the hypothesis. Metric 3 (see Figure 6.3 on page 42) was utilized on the execution of the code generated from the rule *ForRope3* (see Figure 6.19 on page 52) and the four sizes of the unidirectional net (1.44, 1.69, 1.96 and 2.25 million vertices) was used as source graphs. Code was generated utilizing the two versions of the code generator.

The evaluation was performed using 2 cluster nodes. The aim was to find a small interval with a lower and upper bound. The lower bound indicates the highest amount of memory tested which resulted in a failed computation. The upper bound indicates the lowest amount of memory tested which resulted in a successful computation. To

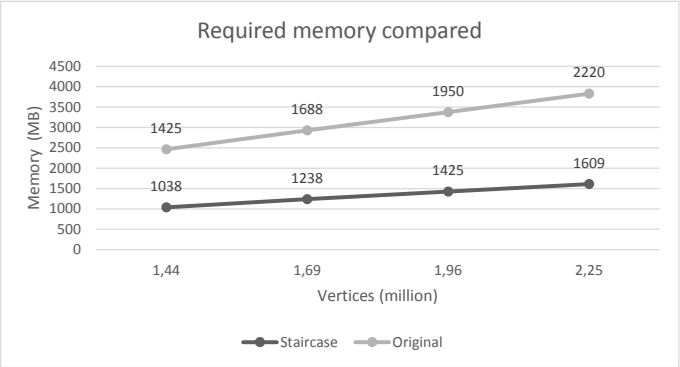
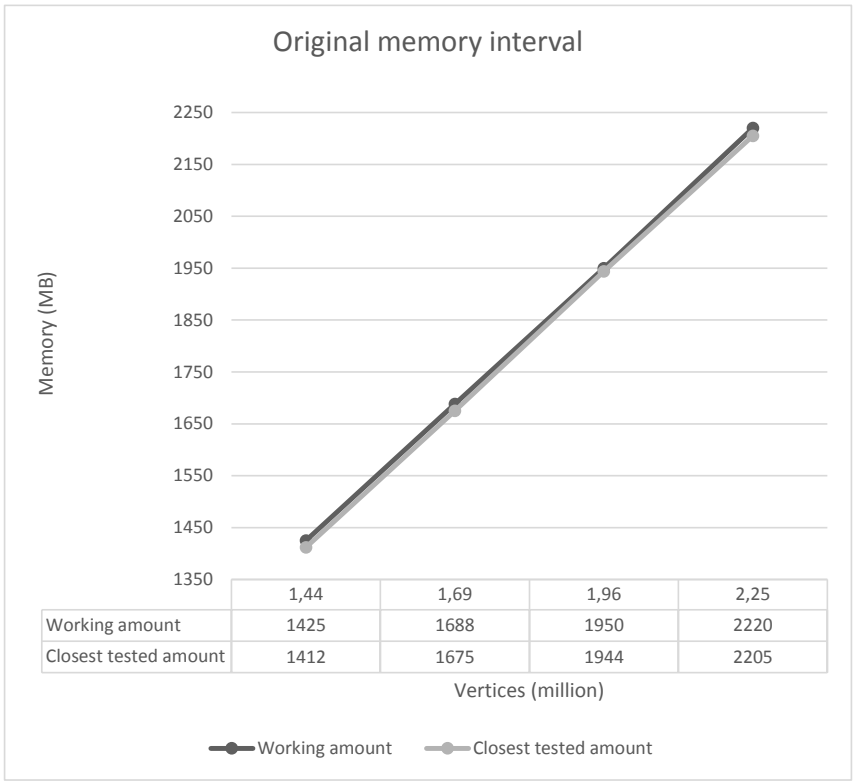


Figure 8.6: Memory consumption for the two implementations compared.

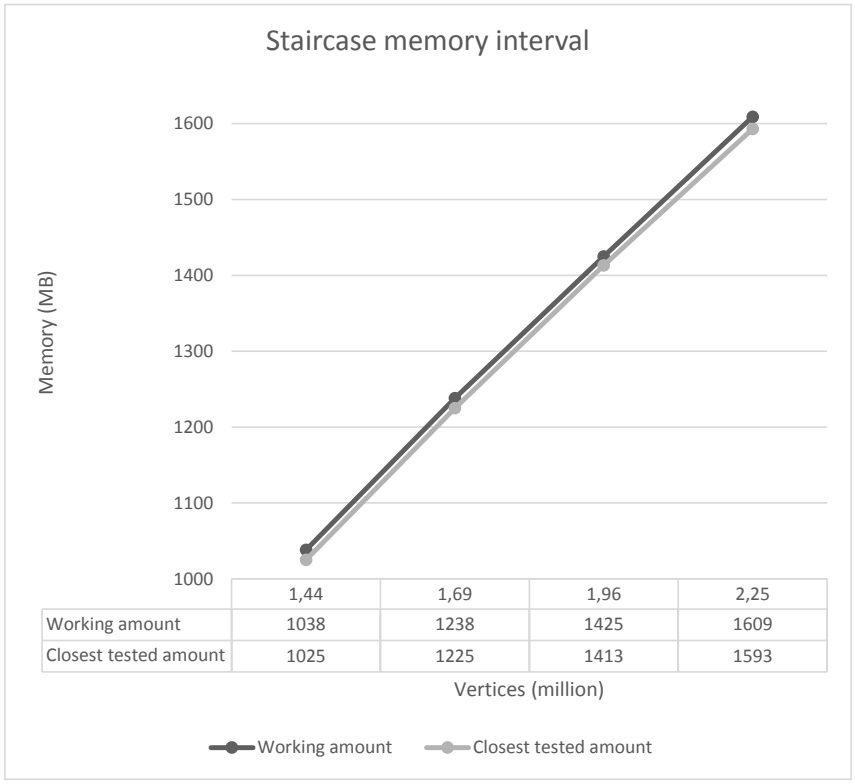
reach this value two values were selected. One which was presumably resulting in a successful execution and one which was presumably not. The computation was then started for the two amounts of memory and the results were checked. A one hour time limit was set for the execution. Binary search was then used to narrow down the interval until it was sufficiently small. The test was performed for the four graph sizes for the two implementations.

The results from the memory tests are presented in Figure 8.6 and Figure 8.7. Figure 8.6 compares the upper bounds for the two implementations when the number of vertices in the graph is increased. The memory required is significant lower for the staircase implementation. It also appears to grow slower when the number of vertices are increased. Figure 8.7a presents the lower and upper bound for the original implementation and Figure 8.7b presents the lower and upper bound for the staircase implementation.

From the data it is apparent that the new solution can deliver a successful job with much less memory than the old solution. It can therefore be concluded that it succeeded in even out the peak of memory usage which is a result from the iterative search plan. However, the test is only conducted for a certain type of rule and on a certain type of graph.



(a)



(b)

Figure 8.7: Working amount and the closest tested amount of memory resulting in a failure for the two versions.

9

Threats to the Validity

As may be expected, this study incorporated a number of threats to the validity. They are sorted according to Wohlin et al [31].

9.1 Conclusion Validity

Threats to the conclusion validity are according to Wohlin et al. [31] threats to draw correct conclusions about the outcome of the experiment and the relation between the two groups. The conclusion validity are further divided in several groups. Of these groups the experiment suffer from *"low statical power"*, *"random irrelevances in the experimental design"* and *"fishing and the error rate"* which are further described below.

In the evaluation using the real data example small samples were used. The reason for this was a lack of available resources for the project. Nevertheless, it is a threat to the validity in the form of *"low statistical power"* [31].

"Random irrelevancies in the experimental design" [31] is also a threat to the conclusion validity of this work. When executing the Giraph computations on the cluster other jobs are executed from other users on other machines in the cluster. These jobs cause noise in the form of network traffic which is a possible error source of the experiment. However when evaluating using the synthetic graphs and rules a high number of executions were run in order to mitigate this validity threat.

In the evaluation multiple comparison were performed. Wohlin et al. [31] described this as *"fishing and the error rate"*. This implies that this work might suffer from the multiple comparisons problem. The problem of multiple comparisons refers to that there is a greater possibility that at least one of the null-hypotheses are falsely rejected when performing multiple statistical test [32].

9.2 External Validity

External validity threats deal with conditions which limits the generalizabilty of the the experiment outside the actual setting [31]. Can it for example be generalized to industrial settings?

There might be an interaction between setting and treatment. The solution was evaluated on a traditional traditional high-performance compute cluster. Such a system is not optimal and designed for Hadoop computations. This might not either be the setting of choice within the industry. For example special software had to be used in order to launch the Hadoop computation on the cluster and the interaction between other concurrent jobs might have an impact on the results.

Further, only the BSP implementation Giraph was used for the experiment. It is possible that the results might not generalize to other frameworks for iterative distributed Graph processing. Also, Henshin was used as the language to define graph transformations. It might be possible that other software tools which are more widely used within the industry have an impact. However the author cannot predict such a case.

Further the evaluation is performed with very specific rules. One of them them is of the special case were joins are present and the other only has edges and vertices of a single type. Therefore it is unclear whether the results generalize to arbitrary rules. However, the sizes has been changed in order to mitigate these validity threats. Both real data and synthetic examples are also used.

9.3 Internal Validity

Threats to internal validity are circumstances which affect the researchers ability to draw correct conclusions about the causality of the experiment [31]. In this setting the evaluation procedure was developed after the problems of the current approach was studied. This can possibly have impacted the design of the experiment in such a way that it focuses too much on the problems of the previous approach and therefore not considers its benefits. However, attempts for mitigating this threat involves the real data example.

10

Conclusions and Future Work

10.1 Conclusions

Model-Driven Engineering is widely used within Software Engineering to increase the level of abstraction and thereby increase the efficiency and maintainability of software project. Combining the notion of Graph Transformations with the Bulk Synchronous Parallel; big data problems can be solved using a model-driven approach. However such solutions are still in their early days and sometimes perform poorly.

This thesis attempted to improve the performance of an algorithm created for mapping and executing graph transformations in the Bulk Synchronous Parallel model. The problems with the current approach were investigated, an alternative solution was designed and implemented, a set of comprehensive benchmarks were developed and the results were evaluated. By doing that, three research questions were attempted to be answered.

> RQ1: How can the performance and memory consumption of the current approach be improved?

By studying and qualitatively accessing the current approach possible problems could be found. These problems were then used in order to design and implement a new solution which aimed to solve the found problems. The new solution built on the idea that the resource requirements of the current solution is not evenly distributed over the computations supersteps. Therefore, a new solution was developed to distribute the resource requirements more evenly over the computations supersteps.

> RQ2: Are the performance and memory consumption improved by the alternative solution?

In order to actually find out if the solution was successful it was evaluated using a benchmark. The benchmark consisted of source graphs, transformation rules and metrics. Using a real data example no significant differences were found however the new solution performed on average worse than the original for the sample data set. Using purposefully developed synthetic graphs and rules the new solution on the other hand performed significantly better when evaluating using two source graphs with different out-degree.

The peak memory usage on the other hand was significantly decreases for the new solution. The value also appears to grow slower when the size of the source graph is increased.

> RQ3: Which properties of the source graph and the graph transformation rule is the alternative suitable for?

In order to investigate if the new approach and the original approach performed varying well for graphs with different kinds of properties the benchmark was developed with this in mind. Therefore, it consisted of real data examples and purposefully developed artificial ones. The results indicate that the new solution might have better performance than the original approach for graph transformation rules executed on source graphs which creates a high amount of partial matches. This is further supported by the the memory tests performed which shows that the new solution can perform computations in environments with significantly less memory.

From the results it is clear that the new solution works better for a certain type of synthetic graphs and synthetic rules. However it is unsure how it works in real data examples. The small evaluation performed using real data did not prove any statistical difference. Overall the new solution did not manage to increase the performance of the current approach in the real data example. A possibly reason for this is the extra overhead caused by it and the failure to manage to handle graphs with vertices not reachable from other vertices, so called joins.

10.2 Future work

This study provided a comprehensive set of benchmarks and a qualitative analyses of the problems with the current approach. All possible problems found are not integrated in the attempt to a solution. Not either all benchmarks are used to evaluate it. Therefore a future study on this subject might use this work as a foundation for further attempts to increase the performance of graph transformations in the Bulk Synchronous Parallel model.

The provided solution is not either completely evaluated for all its characteristics. For example the network usage has not been evaluated because the available environment did not allow that fully. Not either has an optimal amount of *stairs* been investigated.

Further developing the implemented solution is also a path forward. A hybrid solution could for example be developed which utilizes metrics prior to generating the code in order to find the most suitable implementation to use. Such metrics could evaluate the transformation rule properties and characteristics in the source graph. In order to develop such metrics this work can advantageously be used.

Lastly the usage of Bulk Synchronous parallel model as the platform for Graph transformations might not be perfectly suitable for the problem. Other models of computing should also be evaluated. Guo et al. [19] have for evaluated different

graph transformation platforms. Possibly another platform might be more suitable.

Bibliography

[1] C. Krause, M. Tichy, and H. Giese, "Implementing graph transformations in the bulk synchronous parallel model", in *Fundamental Approaches to Software Engineering*, ser. Lecture Notes in Computer Science, S. Gnesi and A. Rensink, Eds., vol. 8411, Springer Berlin Heidelberg, 2014, pp. 325–339, ISBN: 978-3-642-54803-1. DOI: 10.1007/978-3-642-54804-8_23. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-54804-8_23.

[2] S. Kent, "Model driven engineering", English, in *Integrated Formal Methods*, ser. Lecture Notes in Computer Science, M. Butler, L. Petre, and K. Sere, Eds., vol. 2335, Springer Berlin Heidelberg, 2002, pp. 286–298, ISBN: 978-3-540-43703-1. DOI: 10.1007/3-540-47884-1_16. [Online]. Available: http://dx.doi.org/10.1007/3-540-47884-1_16.

[3] D. Laney, "3D data management: Controlling data volume, velocity, and variety", META Group, Tech. Rep., Jan. 2001. [Online]. Available: http://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf (visited on 06/02/2015).

[4] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer, *Fundamentals of algebraic graph transformation*. Heidelberg: Springer, 2006, ISBN: 9783540311874.

[5] The Eclipse Foundation. (2015). Eclipse modeling project, [Online]. Available: http://www.eclipse.org/modeling/emf/.

[6] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer, "Henshin: Advanced concepts and tools for in-place emf model transformations", vol. 6394, Berlin, Heidelberg: Springer-Verlag, 2010, ch. 1, pp. 121–135, ISBN: 0302-9743.

[7] The Eclipse Foundation. (2015). Henshin, [Online]. Available: http://www.eclipse.org/henshin/.

[8] B. Izsó, G. Szárnyas, I. Ráth, and D. Varró, "Incquery-d: Incremental graph search in the cloud", ACM, 2013, pp. 1–4, ISBN: 9781450321655; 1450321658.

[9] L. Valiant, *A bridging model for parallel computation*, 1990.

[10] The Apache Software Foundation. (2015). Apache giraph - welcome to apache giraph!, [Online]. Available: http://giraph.apache.org/.

[11] J. Dean and S. Ghemawat, *Mapreduce: Simplified data processing on large clusters*, 2008.

[12] The Apache Software Foundation. (2015). Welcome to apacheâďć hadoopâő!, [Online]. Available: http://hadoop.apache.org/.

[13] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing", ACM, 2010, pp. 135–146, ISBN: 0730-8078.

[14] The Eclipse Foundation. (2015). Eclipse desktop and web ide's, [Online]. Available: https://eclipse.org/ide/.

[15] ——, (2015). Eclipse modeling - m2t - home, [Online]. Available: https://www.eclipse.org/modeling/m2t/?project=jet.

[16] C. Krause. (). Why model transformations != graph transformations, [Online]. Available: http://www.ckrause.org/2014/08/why-model-transformations-graph.html.

[17] The Apache Software Foundation. (2015). Giraph - implementation, [Online]. Available: http://giraph.apache.org/implementation.html.

[18] A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design science in information systems research", *MIS Quarterly*, vol. 28, no. 1, pp. 75–105, 2004.

[19] Y. Guo, M. Biczak, A. L. Varbanescu, A. Iosup, C. Martella, and T. L. Willke, "How well do graph-processing platforms perform? an empirical performance evaluation and analysis", in *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, ser. IPDPS '14, Washington, DC, USA: IEEE Computer Society, 2014, pp. 395–404, ISBN: 978-1-4799-3800-1. DOI: 10.1109/IPDPS.2014.49. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2014.49.

[20] The Eclipse Foundation. (2015). Emf-incquery: High performance graph search for emf models, [Online]. Available: https://www.eclipse.org/incquery/.

[21] G. Bergmann, I. Ráth, and D. Varró, "Parallelization of graph transformation based on incremental pattern matching", *Electronic Communications of the EASST*, vol. 18, 2009.

[22] G. Bergmann, A. Horváth, I. Ráth, and D. Varró, "A benchmark evaluation of incremental pattern matching in graph transformation", English, in *Graph Transformations*, ser. Lecture Notes in Computer Science, H. Ehrig, R. Heckel, G. Rozenberg, and G. Taentzer, Eds., vol. 5214, Springer Berlin Heidelberg, 2008, pp. 396–410, ISBN: 978-3-540-87404-1. DOI: 10.1007/978-3-540-87405-8_27. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-87405-8_27.

[23] C. Berge, *Graphs*, English. Amsterdam; New York, NY, U.S.A; New York: North Holland, 1985, vol. 6, pt. 1.; 6, pt. 1, ISBN: 9780444876034; 0444876030.

[24] *Ieee standard adoption of iso/iec 15939:2007- systems and software engineering - measurement process*, English, 2009.

[25] S.-Y. Kuo, S.-K. Lu, and F.-M. Yeh, "Determining terminal-pair reliability based on edge expansion diagrams using obdd", *Reliability, IEEE Transactions on*, vol. 48, no. 3, pp. 234–246, 1999, ISSN: 0018-9529. DOI: 10.1109/24.799845.

[26] A. Rauzy, "A new methodology to handle boolean models with loops", *Reliability, IEEE Transactions on*, 2003.

[27] F. Wilcoxon, "Individual comparisons by ranking methods", English, *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945.

[28] A. Vargha and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong", English, *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[29] R. D. C. Team, *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing, 2008, ISBN: 3-900051-07-0.

[30] M. Torchiano. (2015), [Online]. Available: `http://cran.r-project.org/web/packages/effsize/effsize.pdf`.

[31] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering: An Introduction*. Norwell, MA, USA: Kluwer Academic Publishers, 2000, ISBN: 0-7923-8682-5.

[32] R. G. Miller, *Simultaneous Statistical Inference*, English. New York, NY: Springer New York, 1981, ISBN: 9781461381228; 1461381223.