UMEÅ UNIVERSITY
Department of Computing Science
Master Thesis

A Prototype Implementation of

Incremental Millstream Systems

Name
Offer
E-mail offer@acc.umu.se

Supervisor
Suna Bensch

Assistant supervisor
Frank Drewes
# Contents

1 Introduction
2 Background

## Background

2.1 Linguistic background
2.2 Definitions and preliminaries
2.3 Millstream systems

3 Graph Transformation Tools

3.1 AGG
3.2 GraphSynth
3.3 DiaGen
3.4 JGraphX (Java library)
3.5 JUNG (Java library)
3.6 Conclusions

4 Towards an implementation

4.1 Specification
4.2 AGG revisited

5 Incremental Millstream Simulator

5.1 Graph model
5.2 MillSystem and its components
5.3 Rules, configurations and derivation
5.4 Millstream XML file
5.5 User interface
5.6 Graph matching algorithm
5.7 Obstacle-avoiding edges
5.8 System overview
5.9 Test run

6 Conclusions

6.1 Discussion
6.2 Difficulties
6.3 Lessons learned
6.4 Extensibility

A User manual

A.1 Word list
A.2 Starting MillSim
A.3 Creating a MillSystem
A.4 Modifying a MillSystem
A.5 Running a derivation
A.6 Saving a MillSystem
A.7 Exporting graphs or rules
A.8 Shortcuts
A.9 Known bugs and problems

B XSD schema

November 11, 2012
Abstract

This thesis is concerned with a mathematical framework for describing sentences of natural language, called Millstream system. The main goal was to develop a prototype that simulates Millstream systems in an incremental fashion using graph transformations, based on an article [5] mainly written by people in the research group Natural and Formal Language at Umeå University.

The first step was to investigate existing open-source graph tools to see in which extent they could be use for simulating incremental Millstream systems. The conclusion was that none of the existing tools was appropriate for this.

A prototype was therefore implemented from scratch in Java using the JUNG graph representation and visualization library. The implementation, called Mill-Sim, fulfills the goal of a visual non-deterministic simulator for incremental Millstream systems, with import and export capabilities.
1 Introduction

Computer helps us with almost everything. One of the greatest difficulties for computers is the understanding of human languages. Linguistics has built theories to explain the syntactical structure, the semantics and the phonology of natural languages. Traditional Chomsky linguistics has a syntactic central view, which means that everything starts from syntax, and then consider the other aspects such as semantics and pragmatics. Linguistics like Jackendoff [11] promote a parallel architecture, where several aspects like syntax, semantics et cetera works together and are not viewed as a linear order. Computational linguistics tries to formalize languages and build mathematical system describing them.

Researchers at Umeå University presented a non-hierarchical mathematical framework for modeling sentences in natural languages, called Millstream system [3]. It models different aspects of languages simultaneously with independent modules, without any particular linguistic theory in focus as long as they can be described as graphs. The dependencies, or links, between the modules are represented as logical formulas. A Millstream configuration is a full analysis of a sentence, with modules and links according to the formulas. Figure 1 illustrates a configuration.

![Figure 1: The triangles correspond to the modules (often trees) and the links represent the dependencies.](image)

The authors of the first Millstream paper [3] asked themselves how the understanding process could be simulated while reading or hearing a sentence. They proposed a way of incrementally build Millstream configurations [5] word by word. It is uses a lexicon that associates words with graph transformations rules. Roughly speaking, an incremental Millstream system is building a partial structure that grows for each word read, until reaching the last word and the full analysis of the sentence, a Millstream configuration. In order to further analyze and investigate how well this framework works, especially with a larger lexica and more advanced rules, an implementation was wished for.

Incremental Millstream system are until now only theoretical, and no research has been done in order to automatize the rule application in a prototype implementation. The goal of this thesis is to investigate to which extent the freely available graph transformations tools could be used for implementing a prototype. Finally, provide a prototype implementation to simulate the incremental construction of Millstream systems, based on either existing tools or by building one from scratch.

Section 2 describes the preliminaries regarding linguistic and Millstream systems. In Section 3 some graph transformations tools are reviewed, as well as a few libraries that could be used for an own implementation, while Section 4...
states the requirements for a tool and investigates the graph transformation tool AGG more deeply. The decision to create a prototype from scratch was made after these investigations. Section 5 describes the prototype simulator, called MillSim, the representation of graphs and rules, and a few interesting algorithms created during the development. Conclusions and difficulties of the prototype, as well as extensibility, is discussed in Section 6. Appendix A is an user manual for MillSim. Finally Appendix B lists the XSD schema for storing Millstream systems in MillSim.
2 Background

This section describes preliminaries of linguistics, graph theory and Millstream systems, in order to get familiar with the background, basic notions and terminology for this thesis.

2.1 Linguistic background

Linguists often use different aspects when describing languages, such as syntax, semantics, phonology and morphonology, see for example [11]. Syntax is about the principles and rules used when constructing sentences and phrases in a language. The syntactical structure of a sentence is often represented by a tree, see Figure 2. The example (Figure 2-4) is from Jackendoff’s example in [11]. Ignore the subscript numbers for the time being. The sentence \( S_1 \) The little star’s beside a big star can be divided into a noun phrase (NP) and a verb phrase (VP). The NP consist of a determiner (Det), an adjective phrase (AP) with an adjective (A), and a noun (N). The noun also carries information about the person (third) and plurality (singular). This information is carried to ensure that the noun and the verb of the sentence agree in person and plurality (third person singular in this example), and is called a word agreement. Generally a word agreement is a compatibility statement in sentences between word classes and their grammatical categories, such as person, gender or plurality.

The rest of the tree is constructed in the same manner, but also including a verb (V), an inflection (Infl), a propositional phrase (PP) and a proposition (P).

![Figure 2: Syntactical tree of the sentence "The little star’s beside a big star".](image)

Semantics is the meaning of a sentence or a phrase. This can also be represented as a tree, as seen in Figure 3 (even though a general graph would be possible as well). Every node represents a conceptual element, such as a state, event, property or place. The node \( ^{\text{pres,3sing}} \) pres\text{situation}, for example, indicates a present tense situation, namely of a little star being beside a big star.

The node \( ^{\text{state}} \) state is a state-indicator with the verb be (base form of is) which
Incremental Millstream Systems

Background

takes two arguments, in this case an object and a place. The object itself contains three nodes, each giving meaning to the object. The first establishes a mapping from the object into a category – it is a star. The next node \textit{def} (definite) tells that it is a specific object. The third node is a property modifier, which says that the object is small. The other argument of \textit{be} is the place, which has a function \textit{beside} that only takes one object argument (a big star). The rest of the tree is constructed in a similar way.

There is almost always ambiguity in semantics. The two stars could be two real astronomic stars, two drawn stars on a piece of paper, or even two movie stars standing beside each other. The little and big properties could, for example, refer to physical size or how famous the movie stars are.

![Semantic tree of the sentence "The little star's beside a big star".](image)

Phonology is the study of the sounds of spoken language and how it encodes meaning, as seen in Figure 4. One of the main issues in phonology is to group sounds that have the same function in a specific language, and to identify sounds that differ in meaning. In our example the sentence is divided into phonological words, and eventually sub-words and clitics. In Figure 4 for example, /\textit{star}/ /\textit{z}/ (representing the pronunciation of \textit{star’s}) is a phonological word (\textit{Wd}) that consists of the sub-word /\textit{star}/ and the clitic /\textit{z}/. A clitic can in a grammatical sense be an own word or as an affix of a word, as seen in the example where both /\textit{a}/ and /\textit{s}/ are clitics.

![Phonological tree of the sentence "The little star’s beside a big star".](image)

Jackendoff [11], and other contemporary linguists, promote to consider linguistic aspects as autonomous modules that work simultaneously but are linked
with each other through interfaces. These interfaces describe interdependencies among linguistic aspects. Modern linguistics refrains from the syntactocentric architecture of natural language in which all linguistic aspects are derived from syntax [7].

The links are shown as the subscript numbers in each tree in Figure 2, 3 and 4. For example, the node $NP_2$ in the syntactical tree is linked with the node $Object_2$ in the semantic tree. This link represents that the syntactical subject $NP_2$ (the little star) corresponds to the first argument ($Object_2$) of the verb of the sentence, which is represented by the node '$be_{state}$' in the semantic tree. Not all nodes, or even all leaves have a correspondence in another tree.

2.2 Definitions and preliminaries

In order to understand Millstream rules and configurations, which are represented as hypergraphs, hypergraphs are defined in Definition 1.

**Definition 1** (hypergraph). Let $\Sigma$ be a doubly ranked alphabet. A $\Sigma$-graph is a quadruple $(V, E, src, tar, lab)$ consisting of finite sets $V$ and $E$ of nodes and edges, source and target functions $src, tar : E \to V^*$, and an edge labeling function $lab : E \to \Sigma$ such that $rank_{src}(lab(e)) = |src(e)|$ and $rank_{tar}(lab(e)) = |tar(e)|$ for all $e \in E$. The components of a graph $G$ will also be referred to as $V_G, E_G, src_G, tar_G, lab_G$. By $G_{\Sigma}$ we denote the class of all $\Sigma$-graphs.

The algebraic approach to graph grammars was invented in order to generalize Chomsky grammars. The *double-pushout* (DPO) approach was one of the first methods for graph transformations [8].

A DPO rule can be described as tuple $(L, K, R)$ of graphs, where $L \supseteq K \subseteq R$. $L$ is the precondition of the rule, the *left-hand side* (LHS) graph. The graph $K$ is called the *gluing graph*, which is a subgraph of both $L$ and $R$. Everything in $L \setminus K$ will be deleted after applying the rule. $R \setminus K$ is the part that is created. To be able to apply a DPO rule on a graph $G$, not only must $L$ be found in $G$, but the *gluing condition* must be satisfied. This says that all vertices in $L$ must also belong to $K$. When applying a DPO rule to a graph $G$, $L$ must be isomorphic of a subgraph of $G$ in such way that there are no dangling edges when deleting $L \setminus K$ from $L$.

A restricted variant of DPO will be described in the next subsection.

2.3 Millstream systems

The view promoted by the modern linguistic theories, described in Section 2.1, has been formalized by computational linguistics resulting in for example Lexical Functional Grammar [3].

Another formalization are Millstream systems [3, 4], which use tree-generating devices to represent linguistic aspects and logic to represent the interfaces between these linguistic aspects.

Millstream systems are a mathematical framework for language description. A Millstream system consists of an arbitrary but finite number of *modules* and an *interface*. Each module is a tree-generating device (representing a linguistic aspect) and the interface connects two or more of the modules. A module
generates a tree independent of other modules. The way a module generates a tree is of no interest – the module could be a tree grammar, a finite-state automaton or even human input. The interface consists of logical expressions that establish links between the vertices of the generated trees. Any type of logic that can express relations between the vertices of the forest (collection of trees) is allowed.

A valid setting of a Millstream system, called a configuration, is a set of trees (generated by the modules) enhanced with links such that all logical expressions specified in the interface are satisfied.

For more detailed information on Millstream systems the reader is referred to [3, 4, 2].

Incremental Millstream systems

Humans interpret and process (written and spoken) language incrementally. Every word, or part of word, gives us a bit more of information to interpret the whole sentence or sentences. During this process we have ambiguity and different possible continuation. When the last word is read (or heard) hopefully an exact understanding of the sentence(s) is present.

In [5] Millstream system configurations are constructed in an incremental fashion to model this process. For every word (that is read or understood) its syntactic and semantic representation is given. With every following word the representation gets larger until a Millstream configuration for the entire sentence is obtained.

For simplicity, as in [5], only Millstream systems with two modules are considered here, namely syntax and semantics. Another simplification is that all links connect two vertices, one from each tree. A Millstream configuration with these assumptions consists of two trees with links between them. To construct such a configuration graph transformation rules are used. The rules that are allowed will only add new structure to the trees and links between them, the rules never delete anything.

A lexicon associates words with graph transformation rules. A sentence is read in its natural order (normally left to right) and one (non-deterministically chosen) lexicon entry is applied for each word. A common example sentence used is Mary loves Peter. Before any words are read the configuration consist of a startgraph. When reading Mary a lexicon entry is applied and the partial configuration represents Mary in a syntactic way, semantic way, and the association between them described as one or more links. After reading loves the partial configuration becomes larger and now represents Mary loves. Finally Peter completes the sentence and it should have built up a complete configuration.

A configuration in an incremental Millstream system is a hypergraph consisting of a set of ordered and ranked trees with links between them. Hypergraphs were selected for modeling convenience [5]. Since we only work with hypergraphs, hypergraph trees and hyperedges, the hyper-prefix is hereafter omitted. An ordered tree is a tree where the (outgoing) edges have a specific order. Trees are here directed towards the root, in opposite to "normal trees" in computing science.

The edges are labeled from a doubly ranked alphabet, the Millstream alphabet Σ,
which means that every symbol \( e \) has two associated ranking numbers, namely
\( \text{rank}_{\text{src}}(e) \) and \( \text{rank}_{\text{tar}}(e) \) ∈ \( \mathbb{N} \). \( \text{rank}_{\text{src}}(e) \) and \( \text{rank}_{\text{tar}}(e) \) determine the number of sources and targets respectively of an edge labeled \( e \). For every edge symbol in a tree, called a \textit{tree symbol}, the target rank is always 1, and the source rank 0 (i.e. a leaf) or larger. An edge symbol is here denoted as \( \text{Label}(\text{rank}_{\text{src}}) \), for example \( VP(2) \). The links are also ordered edges, labeled from the same alphabet \( \Sigma \). Those symbols, the \textit{link symbols}, have always target rank 0 and in our example source rank 2 (linking two vertices, one per module). The target rank in a Millstream system is therefore either 1 or 0. Edges with a label of target rank 1 are in-tree edges due to that the trees are directed towards the root, and edges with a label of target rank 0 are link edges.

In order to minimize and simplify the visualization of graphs, a condensed representation is used where the labeled hyperedges only are shown as the label below its parent (source). Figure 5 shows a tree in both representations.

In [5] incremental Millstream systems with two modules are considered, with links that are connecting exactly those two modules (\( \text{rank}_{\text{src}}(\text{link}) = 2 \)). The definition allows for more modules, and links within trees as well as links connecting a larger number of modules.

The lexicon associates words with a set of graph transformation rules. A rule is a DPO transformation with injective morphism. A DPO rule consists of the graphs \( L, K \) and \( R \), with the relation \( L \supseteq K \subseteq R \). In incremental Millstream systems a restricted type of DPO rules are used, where \( L = K \). This type of rules never delete anything, and are therefore called \textit{incremental}. A rule, or a \textit{lexicon entry}, can be described as \( L := R \) and be illustrated as the graph \( R \) where \( L \) is marked with a different color. The rule corresponds to the DPO-rule \( L \supseteq L \supseteq R \). Definition 2 defines when a rule can be applied to a graph.

**Definition 2** (rule-applicability). \textit{The rule applies to a graph \( G \) if}

1. \( L \) is isomorphic to a subgraph of \( G \) (for simplicity, assume that the isomorphism is the identity),

2. \( \text{tar}_G(e) \neq \text{tar}_R(e') \) for all tree edges \( e \in E_G \) and \( e' \in E_R \setminus E_K \).
The last statement forbids rules which add new tree edges below a vertex which already has a tree edge. Since the tree symbols in a Millstream configuration always have target rank 1, violation of this statement would lead to unnecessary non-determinism and dead ends.

The idea of an incremental Millstream system is to model the syntactic and semantic processing while reading a sentence word by word, and apply one graph transformation for each read word. This is done by a reader.

**Definition 3** (reader, from [5]). A reader is a quadruple $R = (\Sigma, W, \Lambda, S)$ consisting of a finite set $W$ of words (the input words), a Millstream alphabet $\Sigma$, a mapping $\Lambda$ (the lexicon), and a startgraph $S \in G_\Sigma$. The lexicon $\Lambda$ assigns a finite set $\Lambda(w) \subseteq R_\Sigma$ of rules (the lexicon entries) to every word $w \in W$.

A reading of an input sentence $w_1 \cdots w_n$ by $R$ is a derivation

$$S \xrightarrow{\Lambda(w_1)} G_1 \xrightarrow{\Lambda(w_1)} \cdots \xrightarrow{\Lambda(w_n)} G_n$$

such that $G_n$ is a $\Sigma$-configuration. The set of all $\Sigma$-configurations that result from readings of $w = w_1 \cdots w_n$ is denoted by $R(w)$, and the language (of $\Sigma$-configurations) generated by $R$ is $L(R) = \bigcup_{w \in W} R(w)$.

All incremental Millstream configurations derive from a set of common start-graphs. In the following example the startgraph only contains an edge labeled $S_{(2)}$ with its three connected vertices (one root vertex, two vertices below) in the syntax tree. The lexicon entries for Mary, loves and Peter are shown in Figure 6.
In the sentence 'Mary loves Peter' the first read word is *Mary*, which maps to (the set of) the rule in (a). The whole startgraph is the LHS of the rule, colored blue in Figure 7. After applying the rule and adding the black parts of the figure we have a partial configuration, which also is shown in Figure 7. The next word is *loves*, and maps to the *loves*-rule (b). The LHS of *loves* is a subgraph of the partial configuration, allowing the rule to be applied. Figure 8 shows the partial configuration after applying the rule. The last word is *Peter* and is applied in the same manner, giving the complete Millstream configuration as shown in Figure 9. Note that this is the only order of the words that will end up in a complete configuration (actually the only way to apply any of the rules), due to the LHS of the rules. To allow *Peter loves Mary* corresponding rules for *Peter* and *Mary* would have to be added as $\Lambda_{Peter}$ and $\Lambda_{Mary}$.

![Figure 7](image1.png) Partial configuration after applying the rule *Mary*. [5]

![Figure 8](image2.png) Partial configuration after applying the rule *loves*. [5]

This example is very simplified. It does not represent the semantical information about when it is happening (present tense), nor does the syntactical side divide the word *loves* to a word and an inflection. It does not show the full capability of Millstream systems, since it does not include any non-determinism when selecting rules.

The figures are from [5] with minor changes, and are used with kind permission from the authors.
Figure 9: Complete configuration after applying the rule Peter. [5]
3 Graph Transformation Tools

In this section several graph transformation tools and libraries are reviewed with respect to their purpose, features and usability. The purpose is to investigate candidates for building a Millstream simulator.

First the tools are described in general, then we discuss usability, graph transformation model, visualization capabilities and how well they can be used together with other tools. Two graph libraries are also reviewed for graph representation, visualization and to which extent they can be used for an implementation.

Since we mainly possess programming skills in Java and C++, and libraries are written to a specific programming language, only libraries written for those languages are investigated.

3.1 AGG

The Attribute Graph Grammar System (AGG) is a rule based visual language environment [1, 14, 15, 16], built by the research team Graph Grammar Group at Berlin Institute of Technology. AGG is written in Java and therefore runnable in multiple operating systems and has been developed since 1997.

Graphs in AGG are directed graphs that allow for loops and multiple edges between vertices. All graph objects (vertices and edges) are typed. A type is defined by the user and contains a name, a visual representation (color and shape) and a set of attributes. Attributes can be any Java object and expressions, both from JDK and user-defined classes. This means that an attribute can be everything from numbers and text strings to objects that interact with the user or the Internet.

A graph grammar described in AGG is defined by an optional type graph, a startgraph, a set of rules and possible constraints. The type graph restricts the types of graphs that are allowed within the grammar by defining specific allowed types of vertices and edges, their attributes and which of them can be interconnected. The startgraph is a specific graph to which a rule is applied in the first transition step. The rules are defined by a left-hand side (LHS) graph and a right-hand side (RHS) graph (RHS). Every rule can have a number of negative application condition (NAC) graphs and positive application condition (PAC) graphs. They all have to be satisfied for a rule to be applied on a graph. The vertices and edges of LHS, RHS, PAC/NAC graphs in a rule can be mapped with each other, so they must correspond to the same vertices and edges during application. In Figure 10 the LHS and RHS describe a rule that will add an edge to a new vertex to every vertex. The number before the vertex type name (here 1vertex) indicates the mapping. Taking the NAC into account, the rule will not be applied if the vertex (1) has (at least) one successor. If the host graph was a tree, one leaf would have a successor after one rule application.

Normally AGG will apply rules non-deterministically, but there exist functionality to force a subset of rules to be applied sequentially, and to set a specific priority to a rule or a set of rules. Graph transformations can be performed in debug mode, which lets the user make the otherwise non-deterministically decisions about matching and rule selection.

In the AGG options window it is possible to set which conditions should be checked including injection, dangling and identification. This is sufficient to
model the type of DPO rules used in Millstream systems.

When starting to build a grammar, edge and vertex types must be defined. Creating rules and graphs is done visually by drawing vertices and edges. Mapping is done by clicking on two vertices on different sides, including LHS, RHS, NAC and PAC. The attributes of graph objects are initially defined by a table for each object.

When a grammar is correctly defined it is possible to step-wise apply rules and see the derived graph during transformations including the attributes.

AGG also supports analyzing in different ways, such as checking whether a graph belongs to a language generated by a graph grammar and do critical pair analysis. Such functionality is not in the scope of this thesis.

Some conclusions in the coming sections are drawn by trying the application on a computer with Windows 7 64-bit with JRE 7.0, and the rest from the AGG Manual [13] that is written for an older version and articles written by Gabriele Taentzer [14, 15, 16].
Usability

AGG has a consistent structure, both in the graphical user interface and in the way different functions are implemented. Although the interface has many icons and menu items, which is a sign of a counter-intuitive design, AGG is easy to navigate. Figure 11 shows the GUI during a normal workflow, trying to simulate a Millstream configuration.

Unfortunately unexpected and untraceable errors occur sometimes when trying to apply rules, using type graphs and loading saved grammars.

AGG is open source and some pages on the website indicate that it is still developed by the creators. The source code is partly well-written and divided into packages, but most of the code is in huge classes with large unordered methods and many ill-named variables that makes it even harder to understand.

Linkability

For both import and export the standardized formats GXL and GTXL for graphs and graph transformations are supported.

AGG is written in Java and can be used in one’s own written programs (without the graphical interface) using only the AGG Engine, and can then be seen as a library for graph transformation.

There is no function provided to enable user input during runtime, but it is possible to write a Java class demanding user input on a specific vertex or edge type.

Graph transformation

AGG supports two types of graphs – simple and typed graphs. A simple graph consists of vertices and directed edges that each must be of a type. Every vertex and edge type is associated with zero or more attributes. The attributes are of Java types, so it can be anything from integers to complex data structures, and even own classes. AGG uses the single push-out (SPO) approach, and implements both positive and negative application conditions. In Habel, Heckel and Taeumer [10] application conditions are discussed, and the authors argue that it allows for specific gluing conditions and closes the gap between pure SPO and DPO. Especially they provide proof that the identification condition and dangling condition can be expressed using application conditions, given that LHS is finite. AGG also provides matching dangling, identification and injective condition for the whole graph grammar.

The other type of graphs is typed graphs. They work in the same manner as simple graphs, except that they must obey a structure formalized in an UML-like schema. The schema contains restrictions of the multiplicity of each object type connections. Multiplicity is the number of linked instances, for example how many (minimum and maximum) edges of a specific type are allowed from a specific type of vertex. The total number of allowed vertices or edges of a specific type can also be restricted.

---

\[1\] Every rule needs to be be handled alone by application conditions, allowing rule-specific gluing conditions.
The edges are not ordered, but the algorithms used to present the graphs try to change the graphical structure as little as possible when applying rules.

**Reader and generator**

To be able to let the system automatically generate rules, and allow for meta-rules, it must have the possibility to create rules during execution. Since rules and rule sets are created manually in AGG nothing points to that possibility, even if reading external files is possible.

Rules are matched non-deterministically but with assistance of eventual rule priorities and rule sequences.

**Visualization**

During graph transformation AGG tries to keep the graphical structure as much as possible. In the options window it is possible to apply a layout pattern, which contains information about if vertices should have fixed positions, and how to place edges and vertices. Specific vertices can be set to a fixed position in graphs before or between rule applications.

The graph model of AGG is directed graphs (not hypergraphs). A vertex is represented as a geometrical shape, and an edge as a line with an arrow.

Each type of vertices and edges can have an own color and drawing mode. A vertex type can be assigned a color and a shape, including filled and non-filled rectangle, round rectangle and circle. An edge type can be assigned a color and a line style that can be solid, dashed or dotted, and eventually bold.

Vertices are presented as the shape with its (potential empty) vertex type name inside. The attributes assigned to the vertex can be shown inside the shape, with the standard `toString()` method in Java. This holds for edges as well, except their text is shown in or beside the middle of the edge.

Edges are normally straight lines, but can be manually bended into two lines. The bending is not preserved after a rule application, that is, existing bendings as well as the bendings in the RHS graph become straight lines.

### 3.2 GraphSynth

GraphSynth is a software that works with generative grammars, and the current version has been developed since 2010.

Due to crashes and errors in the program (using Windows 7 64 bit), most conclusions are made from their website and documentation.

**Usability**

After loading any of the given project examples, GraphSynth crashed before applying any rules. It has a very non-intuitive interface, which crashes often in our setup. It has almost no documentation, and there are many fields and controls that are not self-explanatory. The graphical representation is mixed
Incremental Millstream Systems 3 Graph Transformation Tools

Figure 12: GraphSynth interface. Note incorrectly rotated arrow heads on the edges.

with data, which makes it hard to know what actually matters in a graph transformation context. See Figure 12.

The program is open source and is still being developed. The source code contains large dependent classes with large methods, and there are no indications that design patterns have been used.

Linkability

It does not serve any kind of automatization or import/export to common graph or graph transformation formats. The author was not able to find any way of interacting, such as string input.

Graph transformation

The GraphSynth website gives a quite nice overview of the theory behind the system, which includes the desired algebraic DPO approach.

On their website they mention hyperedges (called hyperarcs), but no information about the implementation or how to use it. Another possible way to implement it, is using 2D shape grammar, and having different shapes for normal vertices and linking vertices.
Reading and generator

It is only possible to load/import projects built within GraphSynth, no other formats.

Neither the website nor the graphical interface gives any information how GraphSynth handles non-determinism.

Visualization

Since there has been no success in test graph transformations, the status of creating trees with links that avoids vertices are unknown, but is unlikely. It is however possible to create Bézier curves between vertices.

Only PNG output is available.

3.3 DiaGen

Most of the documentation and usage of DiaGen is together with DiaMeta, which is a meta model framework that is used for specifying visual languages. DiaMeta is built on Eclipse Modeling Framework which makes it complex to work with. Due to problems with compiling and integrating with Eclipse, conclusions regarding DiaMeta are basically based on manuals and the website.

DiaGen can be used by itself and is a system for developing diagram editors according to hypergraph grammars. With a Diagram Language Specification it will generate Java files that (together with the DiaGen classes) works as an interactive diagram editor. The editor recognizes syntactic correctness, even partial correctness, and handle syntactic highlighting. Since the generated files are Java source code it is possible, and required to make it useful, to use classes from JDK to extend the editor with handles, components and calculations.

DiaGen and DiaMeta has been tried on a computer with Windows 7 64 bit and JRE 7.0, and Debian GNU/Linux 6.0 64 bit with JRE 6.0.

Figure 13: A sample program Tree editor build on the DiaGen framework.
Usability

DiaGen uses a meta programming language for specifying the different parts. A part of a working spec-file is shown below.

```plaintext
terminal tNode[1], tChild[2];
circle(a) ==> tNode(a);
arrow(b,c) inside(b,a) inside(c,d) circle(a) circle(d) ==> tChild(a,d)
```

Here vertices and children are defined as terminals. The first rule says that if a represents a circle (graphically), it also represents a vertex. The second rule says that if there is an arrow from b to c (graphically) and the endpoints are in circles a and d, then a is d’s child.

This generates some Java code and gives a base rest of the diagram editor. A sample program that uses this rule (together with many others) can be seen in Figure 13.

Unfortunately the documentation does not match the version of DiaGen that is available and working.

DiaGen is open source and written in Java. Some files are over-commented with automated useless comments, some have German comments in them. The code does not use parameterized types other than to Object which is meaningless. Apart from that, and a few very long classes and methods, the source code is tolerable.

Linkability

DiaGen itself does not export to standardized formats, but since it is fully programmable it allows for own export functions written in Java.

It is possible to create arbitrary Swing components, which makes it possible to give string input.

Graph transformation

DiaGen itself does not provide structure for a tree, graph or even a vertex – neither graphically or in theory. Everything is defined in the spec-file, which makes it possible to implement trees, hyper-edges and arbitrary restrictions to them.

In one way transformation rules are implemented in a generic way which allows enough types of restriction to create DPO rules. The problem is that they are implemented as a static grammar before compiling/generating the DiaGen application.

Since DiaGen is written in Java it would be possible to define one’s own graph classes and use them, but then the benefits of DiaGen would mostly disappear.
Reading and generator

The rules that are used for the automatic transformations in DiaGen are hard-coded in the spec-file, and is not dynamic in any sense during runtime. It would be possible to create a DiaGen-generator that generates a spec-file and run the new DiaGen application. But even then, DiaGen would not interpret the graph correctly, and could not be used as a lexicon editor or even viewer.

DiaGen itself does not support importing rules or other structures. But still, since it is written in Java it is possible to implement it.

This software is not a graph transformation tool.

Visualization

In matter of illustration, DiaGen provides a nice generic way of creating these. With easy syntax you could specify that a vertices children should always be 30 pixels below the parent, and DiaGen will respect that. Rules can change position, color, and even create smooth transitions/animations regarding to all information accessible.

A rasterization (JPEG for example) is easily implemented in Java since the graphics is painted on a standard Graphics2D object. Output in graph formats has to be implemented by traversing the graph.

3.4 JGraphX (Java library)

JGraphX is an open source graph visualization library for Java based on Swing. It is developed since 2001 and is still developed and frequently updated by a private company JGraph Ltd. The JGraphX license is based on BSD but with a termination clause to avoid lawsuits. The same API exists for other languages such as ActionScript and .NET under the product family name mxGraph.

The graph structure model in JGraphX is called mxGraph. This model handles both the graph structure and visualization properties. Graphs contain cells that can be either a vertex or an edge. As default the position (absolute, or relative to its parent) is specified. It is possible to force it to use a specified layout. A layout moves cells according to an algorithm, which will place the cells (vertices and edges) according to it. Different layouts have different settings, such as distance between vertices, angles et cetera.

The model allows all normal graphs, but not hypergraphs. Some models, for example mxCompactTreeLayout, require additional demands of the graph structure to not behave inappropriately.

It is also possible to create groups. This is done by treating a subgraph as a cell, and can therefore have own visualization settings.

To visualize a JGraphX graph in a Java program a mxGraphComponent is provided, which inherits from a Swing’s component and can be added directly into a Swing application. Customization of vertices is done by loading SVG files. Edges are adjusted by a text string containing information such as arrows and color. The syntax and possibilities lacks official documentation.

JGraphX does not provide functionality to draw connected trees beside each
other. It does not support a way of drawing trees or graphs with edges that avoid other vertices.

The model does not separate visualization (position and styles) from the mathematical graph representation. When looking at the source code of JGraphX two classes stand out. The class mxGraph with over 7800 lines of code, and mxGraphComponent with almost 4600 lines of code. This is a clear sign of bad code [12]. Other indications of bad object orientation and code is long methods, using Object instead of parameterized types and frequent usage of constants.

The documentation of JGraphX contains mainly of a JavaDoc, which covers most classes and methods. The user manual is updated, but only gives a very basic examples and descriptions. There is a living community with many answered questions, both for Java specific questions and for mxGraph in other languages.

3.5 JUNG (Java library)

JUNG, Java Universal Network/Graph Framework, is an open source (BSD) java library for modeling, analyzing and visualizing data as graphs or networks. The first release was in 2003 and is still developed, but quite slow, since the latest version was released January 2010. It is mainly written by three PhD students at the University of California, and is maintained by Joshua O'Madadhain from Google, Danyel Fisher at Microsoft and Tom Nelson at RABA Technologies.

JUNG supports directed and undirected graphs, general k-partite graphs, hypergraphs, multigraphs, forests and trees. The graph model is implemented in a hierarchical fashion with interfaces regarding to their structural properties, see Figure 14. For example, the class SortedSparseMultigraph implements Graph (and therefore Hypergraph) and MultiGraph and can be treated like one or more of them, a Tree can be handled as a Forest or a Graph.

![Graph Model Diagram]

Figure 14: The JUNG Graph Model. Dashed boxes represents interfaces, all other are fully implemented.

Both vertices and edges are defined using parameterized types in all parts of...
JUNG. The transformer pattern is used when a specific type is needed. For example, the DijkstraShortestPath takes a Graph of type \(<V,E>\) and a Transformer with type \(<E, ? extends Number>\) which is responsible for transforming the parameterized edge type to a number (float for example). It is possible to use JUNG without using or care about its visualization packages.

To visualize graphs with JUNG a VisualizationServer is used. This inherits from Swing's JPanel which allows it to be embedded in any part of a Swing GUI application and works as a canvas. Prior to the actual drawing all vertices must have a position. This is done by a Layout. JUNG provides several algorithms for positioning the vertices of different types of graphs, such as TreeLayout that can position vertices of a Forest and SpringLayout that can be used on any Graph (but not Hypergraph or Multigraph).

Painting and labeling vertices and edges is handled by the VisualizationServer via pluggable Renderers, and their parameters. The default renderers can handle parameters such as bended edges, standard vertex shapes, label font, text, and their position relative to their vertex/edge. Each parameter is controlled by a Transformer which can have conditional statements to give different vertices/edges different values.

JUNG also supports key and mouse interaction, both to scale and transform, and to change the actual graph.

The source code is well divided into small methods, classes and packages. JUNG makes extensive use of design patterns, and is documented by a short well-written manual, a good API and commented sample programs.

3.6 Conclusions

This subsection contains over-all conclusions, if the tools and libraries can be fully, partly or not at all useful for this prototype.

Tools

The website of GraphSynth conveys the impression that it is suitable for systems using DPO transformations. When testing GraphSynth it had a lot of stability issues. Apart from that, which could be due to our setup, it did not support importing or exporting graphs/rules and was poorly documented. GraphSynth is therefore rejected as a candidate.

AGG is more interesting. It has been used in several articles of Taenzer ([14], [15] and [16]) and reviewed in [1], which indicates that it is a scientifically promoted tool. It has a nice interface and the bugs are documented in the application and most do not affect the normal working flow. AGG does support the type of graph transformations used in Millstream systems.

Finally DiaGen/DiaMeta which has a different approach, since they work as an intermediate step in the development of tools. The output from DiaGen is Java source code, which uses the open source DiaGen libraries. This makes it possible to change everything – making everything possible in theory. Apart from that, DiaGen is especially designed for creating (mouse-) interactive diagram tools and not graph transformation tools. It lacks abilities of add rules on the fly and working with hypergraphs, even though there is a hypergraph representation.
Libraries

Neither of the libraries could be used directly. Both lack support of representing the type of graphs used in Millstream systems, and drawing edges in a vertex-avoiding way. Their features are summarized in Table 1.

JUNG is well-written and due to the good separation of responsibilities it is possible to add new features, including vertex-avoiding edges and separated tree visualization.

The only part JGraphX seems better than JUNG is how living the project is. The latest version of JUNG is from early 2010, while JGraphX has released around 20 updates during this thesis work.

JUNG is still the best choice of library for this prototype. One of the main reasons is that the graph model (and most other parts of JUNG) is built in a way that encourage extensions, which will be needed to build an incremental Millstream simulator from scratch.

Table 1: Features of JGraphX and JUNG.

<table>
<thead>
<tr>
<th>Library</th>
<th>JGraphX</th>
<th>JUNG</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Graph representation and transformation</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hypergraphs</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Typed edges</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Data within edges</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Numbered edges</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Match subgraph correctly</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Concatenate graph with subgraph</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>Graph presentation</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Calculate position correctly</td>
<td>No</td>
<td>Almost</td>
</tr>
<tr>
<td>Draw vertices with data</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Draw straight edges</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Draw vertex-avoiding edges</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>Misc</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sample programs</td>
<td>Very few</td>
<td>Many</td>
</tr>
<tr>
<td>Well structured source code</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Well documented</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Release periodicity</td>
<td>&gt; 1/month</td>
<td>&lt; 1/year</td>
</tr>
</tbody>
</table>

behind the scenes. To learn the meta-language and internals of DiaGen to be able to implement all the missing features and then implement it would exceed the time for this thesis.

Since all three tools are open source, it is in theory possible to implement everything in all of them. In practice it is not, at least not within the time span of this thesis. AGG will be investigated further in Section 4.2 to determine its use as a Millstream simulator.
4 Towards an implementation

This section contains a listing of general specifications an implementation should satisfy, then a revisit of AGG with respect to these required specifications.

4.1 Specification

General idea

In the final implementation of the prototype it should be possible to

- handle two Millstream modules,
- in an intuitive way build up a Millstream lexicon,
- take input strings from the user,
- handle rules in a non-deterministic way,
- step by step apply matching rules from the lexicon, and
- output each step in a clear tree-separated way.

Interesting extensions to the tool would be to

- handle an arbitrary number of Millstream modules,
- see the non-deterministic choices that are made, and
- generate lexicon entries according to meta-rules.

Requirements

For an existing graph transformation tool to be useful in this Master’s thesis for creating a prototype implementation in the given time frame, a set of requirements are defined. All requirements do not have to be fulfilled, but those which are not, are carefully analyzed for potential solutions or workarounds.

The user interface must be clear, logical and intuitive. It should be easy to work with, without unnecessary steps when using the software. Enough documentation to understand the theory behind the software should be provided.

In case changes are required, and to allow for future changes and extensions, the software must be open source. The source code should be well-written and structured. Preferably the software should still be developed, so new bugs are managed.

The type of DPO rules with injective morphism that are used in incremental Millstream systems should be supported. It should forbid rule application that violate the dangling condition, or at least inform the user about it. The graph model must include labeled directed edges. Preferably it should also support hyperedges, or a way to simulate this.

The tool should support graphical string input to interact with the software in a way that a reader in an incremental Millstream system does. If the software is not fully capable of providing all functionality, the software must provide a way of importing/exporting everything needed for the missing steps, in an automizable matter. Export abilities in standard formal (GXL/GTXL or equal) are preferred to allow for further analysis, and use other tools for visualization.
In order to support generation of rules from other software/scripts the software needs the ability to read rules from files, in a documented way. The software must be able to match rules non-deterministically, if several rules would match the current graph and input string. This could be handled either by picking a match randomly or to be chosen by the user. The software should present all other possible matches, or at least indicate their existence.

The graphical trees should be generated next to each other in the software. The software must make it possible to distinguish normal (tree) edges and links, for example, by different colors or drawing modes. The tree links should be drawn so they do not collide with nodes or edges in a way that leads to ambiguity, preferably with curves. Each step, before and after each transformation, from the starting point to a complete Millstream configuration, has to be shown and has to have the possibility to be exported to a file.

4.2 AGG revisited

In this section AGG is reviewed according to which extent it can be used as a prototype for incremental Millstream systems.

Millstream graph emulation

To enforce rule matching with ordered edges as in Millstream systems, let all tree edges have an integer attribute $\textit{order}$. The left-most outgoing edge of a vertex is 1, and increasing to the right. Since the rank of each vertex is known in a Millstream system the $\textit{order}$ of the matching edges is always known. In Figure 15 this is exemplified. The LHS can only match the right-most edge of the graph, since the integer attribute $\textit{order}$ must be equal.

\begin{figure}[h]
\centering
\includegraphics[width=0.7\textwidth]{figure15.png}
\caption{Implementation of ordered edges in AGG.}
\end{figure}

Since AGG does not support hypergraphs the Millstream framework cannot be implemented directly. Hyperedges can be emulated with typed graphs by creating a new vertex type which is called \textit{hyper-edge-vertex} in the following example. Each vertex is replaced by a normal vertex connected to a hyper-edge vertex. The outgoing edges are still attached to the normal vertex, while the incoming edges are attached to the new hyper-edge-vertex. By creating a type graph it is possible to only allow edges between a hyper-edge-vertex and a normal vertex, which will help to maintain the emulated hypergraph structure. A simple example is shown in Figure 16.

The forest structure of Millstream configurations could be enforced by separating in-tree hyper-edges and between-tree hyper-edges (in the following called...
By creating a type graph that only allows zero or one outgoing in-tree hyper-edge to a normal vertex\(^2\), the links are ignored and the remaining parts have a forest structure.

A new type of hyper-edge called hyper-link-vertex, to allow enforced tree structure by type graph or application conditions.

Visualization

All types of vertices (links and in-tree edges) and vertex (normal vertex and possible hyper-edge-vertex) can be assigned an own color and drawing mode, which gives a nice overview. Unfortunately all edges are drawn as straight lines, or as two line segments of which the midpoint must be selected manually.

In Millstream systems each module is an ordered (hyper)tree. AGG does not take order into consideration, even if it is implemented as shown in Figure 15 so there is no guarantee that after rule application the linear order is preserved, or even that the root will end up in the top of the picture.

Since AGG prefers smaller changes in the graphical structure and the trees in the rules are graphically separated, they would probably still be separated.

\(^2\)Millstream systems using a hypergraph tree representation, with the edge direction towards the root, in opposite to trees that are usually used in computing science, where all edges points away from the root.
and in the same order after a few transformations. This does not work if the rules are imported from GTXL files since they normally do not include screen coordinates.

It is not possible to force AGG to avoid that edges are drawn through vertices. Figure 18 shows the best visualization AGG can make compared to how Millstream systems are visualized [5].

![Millstream visualization in AGG.](image)

When AGG is in debug mode it illustrates every rule matching and rule application to the user. Since AGG does not fulfill the visualization requirements for Millstream systems, the graph export functionality is of importance. The standardized formats GXL for graphs and GTXL for graph transformations are supported in AGG [17]. It also provides information about screen coordinates for every vertex in the format. Each step, graph and rule can be exported to GXL/GTXL or JPEG by a shortcut, or every step automatically to JPEG according to the options window.

**Conclusions**

As mentioned earlier, AGG has been discussed in several articles of Taenzer ([14], [15] and [16]) and reviewed in [1], which indicates that it is a scientifically promoted tool. It has a nice user interface and the bugs are mentioned in the application and most do not affect the normal working flow. Unfortunately some bugs when applying rules prevented an acceptable test case for rules of Millstream style. In practice it can only use rules that are created internally, which makes linking with a lexicon generator impossible. It cannot preserve the graphical layout of graphs including order of children during rule application, which is required for a correct visualization of a Millstream derivation. It does not support the reading mechanism used in incremental Millstream systems, since it is not possible to associate a word with a rule or store any information outside the graph. A rule in AGG is always applicable if LHS matches (and no application conditions prevents it).

In AGG it would probably be possible to simulate incremental Millstream systems without changing the source code using many restrictions for each rule and

---

3 Strictly speaking, AGG could simulate MS as well as any other device, since AGG is Turing complete except for the finite memory on a computer.
have an extra graph to represent all information about applied rules et cetera. This would require so much work for each rule, and still not provide a proper graphical illustration of the configuration, that it is not suitable for using it as an incremental Millstream simulator.
5 Incremental Millstream Simulator

Based on the conclusions in Section 3.6 and 4.2 a tool was written from scratch as a Java program, with JUNG as a ground for the graph representation and visualization. Since the purpose was to create a simulator for Millstream systems, the name MillSim was chosen. In this section the internals and structure of MillSim are described and discussed, as well as the XML representation of Millstream systems that can be saved and loaded with MillSim. The limitations and known bugs are listed in the end of the section.

5.1 Graph model

In order to build a tool that works with graph transformation it is necessary to represent all types of rules, configurations and all its components in a suitable way. When working with an object oriented programming language, which Java is, a central idea is to represent as much as possible as objects. The classes, that create the object instances, have a hierarchy to inherit properties and methods.

An important difference between the graph representation in the Millstream papers and the representation implementation is that the implementation does not use hypergraphs. This choice was made primarily to be able to base the visualization algorithms on JUNG:s way of visualizing trees and forests. Details of the differences will be discussed later in this and the next chapter.

Since the type of graphs, regardless if we talk about configurations, rules or start-graphs, are special in Millstream system, we call them all Millstream graphs. A Millstream graph is represented as an object of type MillForest, which inherits from JUNG:s SparseGraph via LinkedForest and JointLinkedForest. This separates the additional functionality that is required for representing a Millstream graph.

JUNG has two edge types, implemented as an enum EdgeType – directed and undirected. Their implementations of forests and trees are built with directed edges, directed towards the leaves. LinkedForest implements the forest structure in a similar way, and using the same Java interface Forest which makes it possible to treat it like a forest within JUNG (provided forest structural algorithms, forest visualization algorithms etc). LinkedForest differs by allowing for undirected edges between vertices within or between the trees, called links. These links corresponds to binary links in Millstream graphs. See Figure 19.

![Figure 19: An example of a LinkedForest. The dashed lines are undirected edges that represent links in Millstream graphs.](image)

The extension JointLinkedForest introduces another term, joint, which enforces a similar structure as of Millstream graphs. A joint is a non-named vertex (the Java toString() method returns the empty string), while any other
vertex, called node, is always labeled. A joint in JointLinkedForest can only have one or zero children (outgoing directed edges). The graph structure also forbids two joints or two nodes to be adjacent to each other. Links are only allowed between joints. See Figure 20.

![JointLinkedForest](image)

Figure 20: An example of a JointLinkedForest. Every second vertex in the hierarchy must be a joint, and links can only exist between them.

The root of a JointLinkedForest could be either a joint or a node, but since the aim is to represent Millstream graphs where a node vertex corresponds to a (labeled) hyperedge, we could assume that the root will always be a joint.

JUNG uses parameterized types \(<V,E>\) for all its graph objects to specify the vertex and edge type as Java classes. LinkedForest and JointLinkedForest are also using these parameterized types, while the MillForest specifies them as MillVertex and MillEdge. MillVertex holds information about module, label and rank. MillEdges only contains the order of the edge, so the children of a node can be sorted correctly. The outgoing edges in MillForest are sorted with this order. Otherwise, a MillForest behaves as a JointLinkedForest with a few helping methods.

### 5.2 MillSystem and its components

A Millstream system, an instance of MillSystem, contains a few components: module names, symbols, rules and startgraphs.

The module names are defined as a non-empty ordered set of non-empty strings. The type of Millstream systems that can be represented (and implemented) are not restricted with only two modules (linguistic aspects). A normal set could be \{SYNTAX, SEMANTIC\}. Note that the order matters only when it comes to visual representation.

The symbols are stored as a set of MillSymbol. A MillSymbol, consists of a module name (see above), a string label and a non-negative integer rank. Note that a MillVertex (the vertex component of a MillForest) inherits from MillSymbol, and can only be created from a MillSymbol. The difference is that every MillVertex is unique, while MillSymbol has the uniqueness defined as the
combination of uniqueness of *module*, *label* and *rank*. A joint is a *MillVertex* with an empty *label*.

The set *rules* is a set of *MillRule* (described later). They are stored as a map from a word to a set of *MillRules*, so the *word* information is actually duplicated since each rule knows its word as well. This was a design choice to reduce dependencies.

**Startgraphs** is just a set of *MillForest*.

### 5.3 Rules, configurations and derivation

A Millstream rule (a lexical entry), an instance of *MillRule*, consists of a word, a left-hand side (LHS) *MillForest* and a right-hand side (RHS) *MillForest*. *MillRule* is responsible for retaining that the vertices of the LHS graph are a subset of the vertices of the RHS graph, in the way implied by Definition 2 in Section 2.3.

A (partial or complete) Millstream configuration is represented by a *MillConfig* object. This has the capability of applying a *MillRule* and return a set of new *MillConfig*s, which is all (possible none) configurations which are derived from the configuration giving that rule.

All derivations are handled by a *MillDerivator*. This is loaded with a Millstream system (*MillSystem*), and contains a *DerivationForest*. A *DerivationForest* is a forest graph where the vertices are *MillConfig*s. The roots of the forest are configurations representing the systems startgraphs. By applying a word to the *MillDerivator* it will use the *MillSystem* and try to apply all rules matching that word. It will try to apply them to each *MillConfig* in the lowest level of the derivation tree. This implies that all non-deterministic derivations are explored, and attached as children on the corresponding leaves in the *DerivationForest*.

This is all happening behind the scenes when using the MillSim GUI.

### 5.4 Millstream XML file

To be able to load and save Millstream system, it is stored as a XML file on disk. It consists of modules, words, symbols, startgraphs and rules.

The XML structure is defined in a XSD Schema definition, see Appendix B. Modules and words are just unique strings of almost any character. Each symbol must have a module, a label and a non-negative integer rank.

A startgraph contains of a set of modules and a tree hierarchy of joint and node. A node requires a label attribute. Joints have an optional id attribute. After the modules is a set of links which must contain two id numbers of previously labeled joints.

A rule contains a word (defined earlier) and a tree in the same structure as a startgraph. The only difference is that each joint, node and link have an optional left-hand-side boolean attribute.

When a XML file is parsed it can complain about missing id-references of links,
forbidden LHS structures and words in rules that are not defined.

5.5 User interface

Currently MillSim only support usage via a GUI. The GUI consists of a scalable window with five tabs – Modules, Symbols, Rules, Startgraphs and Derivation. See Figure 21. The Millstream system is defined by the first four tabs. The Derivation tab is using the information from the defined Millstream system to apply rules and build a derivation forest. The derivation forest consists of vertices, which represents each startgraph (the roots) or a graph derived from a startgraph (everything below the roots). All graphs can be exported to PDF, normal image and TikZ latex-code.

Figure 21: The GUI. Here the defined symbols are listed.

![Figure 21: The GUI. Here the defined symbols are listed.](image1)

In the bottom of the window the status bar shows any notification or error occurring during the usage of the program. In the File-menu it is possible to save or load a Millstream system.

Figure 22: The derivation step "loves" is selected from the derivation forest, with the MillForest corresponding to the partial configuration to the right.

![Figure 22: The derivation step "loves" is selected from the derivation forest, with the MillForest corresponding to the partial configuration to the right.](image2)
5.6 Graph matching algorithm

The largest algorithmic problem within this thesis was to correctly match a subgraph, to be able to apply rules. In general subgraph matching is NP-complete, but for special types of graphs such as planar graphs it is possible in polynomial time [9]. One could easily argue that the graphs within real MillStream systems will not be very large, since their purpose is to represent a sentence. The main focus when developing was to have a working algorithm, rather than an efficient.

The current graph model in MillSim only allows for forests, but the following algorithm is created to support more general graphs, directed acyclic graphs (DAGs), in case of future development.

Algorithm 1 Find matchings of subgraph $H$ in $G$.

**Precondition:** Directed acyclic graph $G$ and $H$.

1. function \text{MatchSubgraph}(G, H)
2. \hspace{1em} Set $\langle v_h \rangle R \leftarrow \text{Roots}(H)$
3. \hspace{1em} Map $\langle v_h, Set \langle v_g \rangle \rangle \leftarrow \text{VertexMatchings}(G, R)$
4. \hspace{1em} Set $\langle Map < v_h, v_g > \rangle Pr \leftarrow \text{Permutations}(Mr)$
5. \hspace{1em} Set $\langle Map < v_h, v_g > \rangle M \leftarrow \{\}$
6. for all Map $\langle v_h, v_g > p \in Pr$ do
7. \hspace{1em} Map $\langle v_h, v_g > m \leftarrow \text{DAGMatching}(G, H, p)$
8. \hspace{1em} if $p$.size() = 0 then
9. \hspace{2em} Continue
10. \hspace{1em} if MatchLinks(V, G, m) then
11. \hspace{2em} $M \leftarrow M + m$
12. return $M$

The idea is to find all ways of matching the roots of $H$ (Roots and Permutations) in $G$, represented as a set of mappings from vertices (roots) in $H$ to vertices in $G$. Matching means that the vertices correspond to a MillVertex from the same MillSymbol (either a joint or a matching node). Each root mapping is then extended, by traversing $H$ and $G$ simultaneously and find matchings.
Corresponding links in $G$ (MatchLinks).

MatchSubgraph will retrieve the roots of $H$ by calling Roots($H$). Roots iterates over all vertices in $H$ and returning the ones without a predecessor.

The function VertexMatchings, see Algorithm 2, matches all vertices (roots of $H$) to a set of vertices of $G$. For each root it iterates over all vertices in $G$. If some root does not match to any vertex in $G$, it is not possible to find a (full) matching. Exceptions and details of this level are omitted in the algorithm descriptions, in order to offer a better overview. When all roots are matched, the matchings are returned as a map from each root to a set of vertices in $G$.

Algorithm 3 Returns the set of all permutations (1-to-1 maps) given a 1-to-several map.

Precondition: A mapping $M$ from an object to a non-empty set of the same type of object.

```plaintext
1 function Permutations(M)
2   Set < Map < v_H, v_G >> res ← {{}}  // The set of one empty mapping
3   for all (v → S) ∈ M do
4      Set < Map < v_H, v_G >> newRes ← { }
5      for all Map < v_H, v_G > r ∈ res do
6         for all v_0 ∈ S \ values(r) do
7            newRes ← newRes + copy(r).put(v, v_0)
8      res ← newRes
9   return res
```

Instead of a mapping from each root to a set of vertices in $G$, Permutations returns all permutations of roots, which means all possible combinations of root matchings. The algorithm starts by creating a set of one empty mapping, which will be the result set. For each vertex, the result set will be duplicated once for each matching of the current vertex. It will skip vertices that already has been matched. The result set can easily be large even for a small example. For example, if each vertex of three roots of $H$ is matched with two vertices in $G$, the algorithm will return a set of $2^3 = 8$ root mappings (permutations).

Finally DAG Matching, Algorithm 4, is called to try to find a full matching of the subgraph $H$ in $G$ given one of the root permutations, by growing the DAGs simultaneously and match the vertices. It does this by incrementally building up a map one root at the time with OneRootDagMatching.

OneRootDagMatching is a recursive function that with a depth-first search of $H$ from a given vertex (roots initially). If the vertex has children in $H$ it will try to match these with the corresponding children of $G$ (FullMatching). To ensure that the children mapping does not interfere with the larger mapping, this is checked (MisMapped).

The subfunctions Roots, MatchLinks, FullMatching and MisMapped are not described since they do not contain anything essential, nor do they affect the complexity of the algorithm.

To be able to discuss complexity of this algorithm some complexity values are stated. In general, when investigating graph complexity, one of the most important values is the number of vertices. Further on we will use the absolute
Algorithm 4: Trying to find a matching of $H$ in DAG $G$ given a root vertex mapping $R$.

Precondition: DAG $G$ and $H$, and a mapping $R$ from each root in $H$ to vertices in $G$.

1 function DAGMatching($G, H, R$)  
2 \hspace{1em} $Map < v_H, v_G >> M \leftarrow R$  
3 \hspace{1em} for all Vertex $r \in R$ do  
4 \hspace{2em} if not OneRootDagMatching($G, H, M, r$) then \hspace{1em} $\triangledown$ Will extend $M$ with vertex matchings  
5 \hspace{2em} \hspace{1em} return NULL \hspace{1em} $\triangledown$ Subgraph not match $\Rightarrow$ No match  
6 \hspace{2em} return $M$  

7 function OneRootDagMatching($G, H, M, r$)  
8 \hspace{1em} Set $< v_H > suc_H \leftarrow H.successors(r)$  
9 \hspace{1em} Set $< v_G > suc_G \leftarrow G.successors(M.get(r))$  
10 \hspace{1em} if $suc_H = 0$ then  
11 \hspace{2em} return true \hspace{1em} $\triangledown$ No more successors in subgraph $\Rightarrow$ Match  
12 \hspace{2em} Map $< v_H, v_G > M_2 \leftarrow$ FullMatching($suc_H, suc_G$)  
13 \hspace{2em} if MisMapped($M, M_2$) then  
14 \hspace{3em} return false \hspace{1em} $\triangledown$ Inconsistent mappings $\Rightarrow$ No match  
15 \hspace{2em} $M \leftarrow M + M_2$  
16 \hspace{2em} for all Vertex $v_H \in suc_H$ do  
17 \hspace{3em} if not OneRootDagMatching($G, H, M, v_H$) then  
18 \hspace{4em} return false \hspace{1em} $\triangledown$ Subgraph not match $\Rightarrow$ No match  
19 \hspace{2em} $M \leftarrow M + S$  
20 return true
value notation for the size of a set (i.e. $|A, B, C| = 3$). In this context the number of vertices of $G$, $n$, and roots of $H$, $r$, is interesting. The relation $|\text{vertices}(G)| = n \geq |\text{vertices}(H)| \geq |\text{roots}(H)| = r$ should be kept in mind.

A Millstream graph will mostly consist of trees, only by way of exception will it contain shared subtrees (creating a DAG structure which is not a forest). Since the trees represent a linguistic aspect, and a vertex is an atomic element in that aspect, the degree of $G$ (and $H$) is said to be limited by a number $d$ which is the maximum branching a linguistic atomic element can have when represented as a tree. In examples of incremental Millstream papers published so far [15] and [13] this number is 2, but could raise to 3 or 4 in more advance examples. The graph $H$ will be the left-hand-side of a rule. A rule represents one type of occurrence of one word. It is therefore safe to assume that the height of $H$ will be limited by a number $h$. In Millstream papers the maximum height of a LHS is 5 (represented as a non-hyper graph, as MillSim does). The complexity variables are summarized in Table 2.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>$</td>
<td>\text{vertices}(G)</td>
</tr>
<tr>
<td>$r$</td>
<td>$</td>
<td>\text{roots}(H)</td>
</tr>
<tr>
<td>$d$</td>
<td>$\max(\text{degree}(\text{vertices}(G)))$</td>
<td>The maximum degree of $G$</td>
</tr>
<tr>
<td>$h$</td>
<td>$\text{height}(H)$</td>
<td>The height of $H$</td>
</tr>
</tbody>
</table>

Table 2: Complexity variables used in analysis of the subgraph matching algorithm.

For calculating the complexity of \texttt{MatchSubgraph} the subfunctions are considered. \texttt{Roots} iterates over all vertices, with a complexity of $O(n)$. \texttt{Vertex-Matchings} iterates over all roots of $H$, and for each root it iterates over all vertices in $G$, which gives a complexity of $O(r \times n)$. The complexity of \texttt{Permutations} is implied by the fact that it can be a maximum of $n^r$ permutations, giving a complexity of $O(n^r)$. \texttt{DAGMatching} could in worst case traverse the whole tree $H$, with complexity $d^h$ and for each vertex search over $d$ children in $G$. The function will be called once for every root permutation, which is the heaviest part of the algorithm. The total complexity of the \texttt{MatchSubgraph} algorithm is therefore $O((n^r) \times (d^h))$. 
5.7 Obstacle-avoiding edges

In order to visualize a Millstream graph (configuration or rule) in a decent way, the links must be considered. They exist between two joints (unlabeled vertices) normally in two separate trees. The usual way of displaying edges in JUNG is with straight lines between the vertices, or with equally bended lines. In Millstream graphs both vertices (except for the endpoints) and other links should be avoided as much as possible. In MillSim this is implemented by the class \texttt{ObstacleAvoidingEdgeShapeTransformer}.

JUNG visualizes graphs by first deciding the positions of each vertex (in MillSim done by a modified version of \texttt{TreeLayout}), then render the vertices and edges with shapes, colors and eventually text. All these appearances in JUNG are defined by transformer objects (derived from the Apache Commons library) applied to a \texttt{RenderContext} in a VisualizationViewer, responsible for drawing JUNG graphs. Transformers are objects that transforms one object into another, for example a \texttt{Transformer<String,Integer>} would transform a string to an integer, in some way defined in the class. A \texttt{Transformer} object can be as simple as always return the same thing, be dependent on the input (normally), or even communicate with other objects and calculate the output from that.

To configure the shape of edges a \texttt{Transformer<Context<Graph<V,E>,E>,Shape>} is needed. A \texttt{Context} is a container object which in this case contains a graph (with parameterized types \(V\) and \(E\)) and a specific edge (of type \(E\)). With a transformer applied to a Millstream graph \(V\) would be \texttt{MillVertex} and \(E\) \texttt{MillEdge}, but the \texttt{ObstacleAvoidingEdgeShapeTransformer} is generally defined.

When creating a \texttt{ObstacleAvoidingEdgeShapeTransformer} it must be prepared with the a set of vertices and their positions. In MillSim this is done by retrieving the positions from the modified \texttt{TreeLayout} object.

The main idea is to calculate a number of candidate shapes (symmetric quadratic parametric curve segments) for each edge, and scoring them depending on how close they are to \textit{punishing points}, and returning the best curves. The punishing points are all other vertices (except for the endpoints) and points on earlier calculated curves. This implies that the algorithm will get better results after calling it for each vertex once before using the results. To concentrate calculations to the beginning and avoid getting stuck between different shapes, it will not recalculate more than once if the graph has not changed.

The algorithm for calculating is complex, so many details are omitted. The algorithm will prefer to be close to two points rather than very close to one. It will try to avoid the center of the graph, so rather creating curves that are straight, it will bend them outwards. It can be fine-tuned by changing the nine constants defined and documented in the class. Some of them will change the speed of the algorithm significantly.

The \texttt{ObstacleAvoidingEdgeShapeTransformer} can be used on any edge on any graph, but is tested and tuned for links between vertices in an otherwise forest structured graph.

Note that the TikZ graph export functionality in MillSim does not fully use the output of the \texttt{ObstacleAvoidingEdgeShapeTransformer} due to representation differences.
5.8 System overview

Packages

MillSim is organized into a few packages:

- **millsim.graph** is the Millstream graph and rule representation package. It is highly dependent of the JUNG library. It also holds a class for representing a Millstream system and configuration, with an API for applying rules non-deterministically.

- **millsim.graph.util** contains some graph and tree functions not included in JUNG, as well as the graph matching algorithm.

- **millsim.graph.visualize** holds JUNG extensions for visualizing MillForests and MillRules in a desirable way, as well as classes handling mouse behavior for editing this kind of graphs.

- **millsim.gui** contains all GUI classes of MillSim.

- **millsim.helper** contains some static helping functions mostly for debugging and for cheating the JUNG decorator pattern.

- **millsim.io** is responsible for importing and exporting Millstream systems, including graphs and rules. It has a XML reader and writer, as well as a LaTeX TikZ exporter.

- **millsim.main** holds the main class, which only starts the GUI.

Libraries

The main library used in this prototype is **JUNG** which both graph representation and visualization is built upon, see Section 3.5.

Apache’s **log4j** is used for all logging. It has different logging levels to be able to filter the logging, see below.

\[ \text{TRACE} < \text{DEBUG} < \text{INFO} < \text{WARN} < \text{ERROR} < \text{FATAL} \]

Via a config file, `logconfig`, the output to console and logfile can be adjusted by setting a lowest logging level per package or class, without recompiling MillSim. This is useful for detecting errors or unexpected behavior in the program, for example when loading an invalid XML file or trying to create an invalid graph or rule.

The exporting capabilities are handled by the library **FreeHEP** which exports the Swing panel to raster graphics (JPG, PNG etc) and with an extension even to vector-based PDF. MillSim also extends the exporting dialog with an option of LaTeX TikZ code, special designed for MillForests and MillRules.

**JUnit** was used for testing during development of MillSim, especially for the graph representation and algorithms.

All libraries are automatically downloaded and integrated via **Maven**, since they are defined in the `pom.xml` file in the project’s source code root directory.
5.9 Test run

In the following example three sentences will be derived, 'Run', 'Mary runs fast' and 'Mary runs over the hill'. The main focus is to test how derivation works in MillSim. For information about how to create the systems see Appendix A.

![Figure 23: The Derivations tab before applying any words, showing only the startgraphs.](image)

![Figure 24: The rule 'Run' has been applied, giving a very small complete configuration.](image)

The first example sentence only contains the imperative form of 'to run', 'Run', and is the demand or order to run. The word 'Run' is entered in the text box and the button Apply word is pressed. MillSim is now looking up all rules associated with that word, and trying to match their LHS to the startgraphs, and updating the derivation forest and active graph (right side of the window) accordingly. This is indicated by the status-bar in the bottom of MillSim. There was only one way to apply the set of rules associated with the word, giving one of the startgraphs in the derivation forest a child. See Figure 24.
MillSim does not indicate (or even know) that this is a complete configuration, but one can understand that from the fact that all leaves are labeled. In a hypergraph representation this would correspond to that all leaves has a (labeled) hyperedge of rank 1 (only connected to one vertex).

Figure 25: A partial configuration after the first rule ‘Mary’ has been applied.

A slightly longer sentence is ‘Mary runs fast’. The derivation is reset, and after the first rule is applied the other startgraph is extended with one configuration. See Figure 25. Note that the second children of $S$ is still open, which indicates that it is only a partial configuration.

Figure 26: After ‘runs’ applied two possible, and randomly one of them is selected.

The next step is to apply the word ‘runs’. This word applies to two rules, which both are applicable in the current partial configuration. This is clearly shown
in the derivation forest (Figure 26) where two children have been appended to the 'Mary' vertex.

Even though only one configuration is selected in MillSim, both 'runs' configurations are active behind the scene. When applying the next word 'fast', it will try to match the LHS of the rule associated with the word to all configurations in the lowest level of the derivation forest. This application yields the complete configuration, shown in Figure 27.

![Derivation Forest for 'Mary runs fast'](image)

Figure 27: The complete config for 'Mary runs fast'.

The last example is 'Mary runs over the hill' and starts with the same two applications (see Figure 26). When applying 'over' the other branching will die. If the user would try to apply 'fast' now it would not be possible, since the
'over' partial configuration is the only on the lowest level. After applying 'the' and 'hill' as well, the derivation forest looks as in Figure 28.
6 Conclusions

Let us first have a quick look back at the initially requirements of this work. The following list shows which of the requirements, formulated in Section 4.1, have been fulfilled.

- Handle two Millstream modules
- Create Millstream lexicon intuitively
- Take input string from the user
- Handle rules non-deterministically
- Step-by-step matching of rules
- Output each step in a clear tree-separated way

Regarding the extensions discussed in Section 4.1 the situation looks as follows.

- Handle arbitrary number Millstream modules
- See the non-deterministic choices made
- Generate lexicon entries according to meta-rules

In the rest of this section we discuss the use, difficulties and possible extensions of MillSim.

6.1 Discussion

In Section 4, the decision to write a prototype from scratch rather than using AGG was made. This was due to the fact that the tools available were far from satisfactory. AGG could not visualize Millstream graphs in any pleasant way. To build a new visualization system it would require an inexcusable amount of time. It was not possible to identify rules by a word (lexicon entry), which would be required for a Millstream simulator. The source code of AGG is not well separated or organized and is not suitable for further extension, which would be required. The only reasonable choice was to write a prototype from scratch, using an existing visualization library (JUNG).

Regarding the graphical user interface (GUI), which was one of the most time consuming parts of the implementation, a suggestion could be to write a command line tool instead. The main reason for building a GUI was to be able to create the rules. The derivation forest and all partial and complete configurations could easily be exported into images and/or XML files instead. The graph creation could for example be done in AGG or another tool in a specific way, and then imported into MillSim for parsing. This parsing algorithm would probably be implemented much faster than all time spent on creating a working GUI, giving more time for extending the simulator in other ways, see Section 6.4.

Unit testing was used during the development of MillSim. This required the creation of a library of trivial and non-trivial examples of all kinds of situations. This took a lot of time, but hopefully implied a more solid and reliable implementation.

One problem to use this prototype on larger systems would be that the subgraph matching algorithm is inefficient (see Section 5.6). This could be improved in several ways, especially by investigating more than the roots as a first step to eliminate combinations (and permutations) that not could lead to a matching.
The usefulness of the system will still have to be established, but it has actually already been successfully used in one small instance. MillSim is able to parse example sentences like 'Mary loves Peter', 'They sit on the sofa', 'Mary runs fast', 'Mary runs over the hill' and 'Peter sits on the sofa'. This example sentences handle subject-verb agreement (They sit' not 'They sits') and subcategorization (i.e., transitive, intransitive verbs). But even in this small scale, MillSim is also able to parse syntactically correct sentences like 'Mary sits over the hill' that semantically do not make any sense. This shows the need for semantic filters (people can only sit on an object) and better understanding of the semantics for Millstream systems in general.

What speaks against any extensive use of MillSim is that the number of researchers working with, or even knowing of, Millstream systems is quite low. The fact that MillSim does not support the new type of rules defined in the latest technical report [6] does not help. See also Section 6.4 for a discussion.

6.2 Difficulties

This project shows the huge step between an idea based on a formal description and an implemented prototype, and the importance of an unambiguous definition. During the development several problems arised, regarding how the simulator should work and act in some situations. One of the misunderstandings was that the set of joints a link is connected to is ordered, which is not the case in MillSim. Another confusion creating extra work was when it became clear that a rule could match in different ways on the same partial configuration. The derivation algorithm had to be changed because of this. The unintentional restrictions made due to misunderstandings and misinterpretations has not shown to restrict any actual examples of Millstream systems so far.

Most coding issues that occurred during the project concerned the JUNG visualization system and Java GUI programming in general. Despite of much time and energy spent on debugging the GUI, there are GUI bugs, such as not refreshing the graphs when appropriate and incorrectly transform the drawing area when resizing the window.

JUNG seemed to be a well-tested trustworthy graph library, both for representing graphs and visualizing them, and assumed to work correct. This lead to time-consuming debugging two times, when the actual bug was in JUNG. Both bugs were resolved by extending the invalid classes.

6.3 Lessons learned

If the whole implementation should be redone, some other choices could be made for a more efficient and better product.

A full understanding of the JUNG graph model and the visualization system is required for making the best decisions when building a JUNG based tool. Lack of deeper knowledge easily leads to dirty solutions, and unnecessary coding. One should also avoid assuming that even a widely used library is implemented correctly.

Instead of representing links as undirected edges, the links could have a data structure of their own, for example a list. This could be added as an extension to any graph, and would not interfere with the original graph structure. This
representation would also allow for ordered links and links with arbitrary number of joints.

If the LHS would still be a subgraph of the RHS (which is not the case in the newer Millstream paper [6]) a more pliable representation would be to represent it as a graph and a set of vertices noting the LHS.

Even though unit testing was used during the prototype implementation, it was not pure test driven development, since the tests were not written before the actual code. In a new implementation, especially if it was going to be more than a prototype, unit testing would be used during implementation of both representation and algorithms regarding graphs and rules.

6.4 Extensibility

During the development of this prototype, a new article regarding Millstream systems [6] was written. In that article the type of rule has changed, so the LHS is not necessary a subgraph of the RHS. It has introduced nonterminal hyperedges. Extending MillSim to support this kind of rules would require significant changes not only in representation and the rule application algorithm, but also the user interface for creating rules. All other parts of MillSim could be kept, as well as the current subgraph matching algorithm.

To go from the current forests into a DAG structure of startgraphs and rules, there are several ways to go. The simplest, not a completely sound solution, would be to mark all edges that violate the forest structure. This would allow for only minor changes in the vertex positioning class TreeLayout (provided by JUNG) to ignore the marked edges and treat the graph as a forest. The XML representation could take the same approach by using the hierarchical structure for the forest, and represent non-tree edges in a similar way as the links separately with vertices identified by their id. The subgraph matching algorithm could remain the same, and only small changes in the rule application algorithm. A clean solution would require a new, probably complex, algorithm for visualizing the graph. Not only must it visualize the DAGs satisfactorily, it should order the edges from left to right since the order matters in Millstream system (and sentence representation).

After creating a Millstream system in MillSim all derivations are tested manually. An interesting extension would be to see all possible rules that could be applied in a specific state in a derivation. This could be used for analyzing a built system in order to find flaws (possibility to derive incorrect sentences) without trying all possible words. This extension would only require small changes – instead of only trying to match the rules associated with a specific word, it should try all rules in the system. This could also be done automatically in several steps, giving a full representation of all sentences derivable from the system. In that case some mechanism to detect infinite derivations should be implemented as well.

An important question for future research is to generate lexica (set of rules) in a consistent manner [3]. This could be done in a separate program/tool, and imported to MillSim via the well-defined XML structure. A suggestion would be to use MillSim as a library for creating such a tool, since representation and XML import and export already is implemented. The step to later integrate this entirely into MillSim would be small.
As mentioned MillSim can be used as a library, since it is divided into classes and packages, each with own responsibilities. To extend MillSim to a command line tool would not require much effort. The input could be a XML file and a sentence, and the output a XML file with configuration(s) or a set of images.
References


A User manual

A.1 Word list

Within MillSim a specific terminology is used.

- **MillSim**: The prototype application.
- **MillSystem**: A tuple of modules, symbols, rules and startgraph. Loaded in MillSim or saved as an XML file.
- **MillForest**: The type of graph representing a forest with joints and nodes, with possible links between joints.
- **Vertex**: A specific vertex in a graph, either a joint or a node.
- **Joint**: A specific non-labeled vertex. The type of vertex that can have connected links in a MillForest.
- **Node**: A specific labeled vertex. The type of vertex that can have more than one children. Nodes are labeled from, and created by, a symbol.
- **Symbol**: A tuple of module, label and rank within a MillSystem. Can instantiate nodes.

A.2 Starting MillSim

MillSim is distributen as an executable JAR file, called millsim_build-xxxxxxxx.jar, where the x:s is the build number. The latest version is available at [http://www.acc.umu.se/~offer/millsim](http://www.acc.umu.se/~offer/millsim). Java Runtime Environment (JRE) 1.7 (or higher) is necessary to execute MillSim, which is available for most desktop operating systems.

Run MillSim by execute the file, or by typing `java -jar millsim_build-xxxxxxxx.jar` in a terminal.

If the JAR file is placed where the user have write access it will create a logfile, `millsim.log`. By running MillSim from a terminal equivalent information will displayed there (`stderr`).

A.3 Creating a MillSystem

The user must follow a specific order when creating a new MillSystem (representing a Millstream system).

When starting MillSim, or selecting New MillSystem in the File-menu, the user is presented an empty list in the Modules tab. See Figure [29] The user should now add module by typing their unique names in the text box and press Add. The module name must be a string of any letter, space, hyphen, underscore and normal parenthesis. The modules should be added in the way they should be presented visually, from left to right.

**Note**: Modules should not be removed when there exist symbols with that module. This could lead to unexpected behavior.
The next step is to add the symbols. Switch to the Symbols tab. New symbols are added by right click on any empty space and select Add symbol. Select a module from the list. Type a label which must be a string of any letter, space, hyphen, underscore and normal parenthesis. Type a non-negative integer as a rank. The rank represents the number of child that the symbol will have. If these three parts are consistent, the consistent check-box will be checked, as seen in Figure 30. All symbols must be unique. Uniqueness is defined as an unique combination of module, label and rank. To get a better overview the symbols can be sorted by any column by clicking on the column header. Symbols are removed by right click on the symbol and select Delete symbol.

**Note:** Symbols should not be removed when there exist rules or startgraphs with that symbol. This could lead to unexpected behavior.

![Figure 30: MillSim indicates the consistency of each symbol.](image)

Now rules and startgraphs should be created. These steps can be done in any order.

A rule is created by switching to the Rules tab and clicking on the text area to the lower left corner. Write the word that should be associated with the new rule. A word must be a string of any letter, space, hyphen, underscore and normal parenthesis. Normally it is just one word. Click New to create the rule, and an empty space will occur on the large middle area of the GUI.

A graph is created by left click to add vertices, and click-and-drag between two vertices to create an edge. To add a vertex a symbol must be selected. Select
a symbol from one of the modules listed to the right. The first symbol in each module is named `MODULE Joint` and is used to simulate the hypergraph used in real Millstream graphs. The rest of the symbols can be selected to create nodes of that type, which are defined in the `Symbols` tab. Click on the empty space to add vertices of the chosen type. To delete vertices, right click on it and select `Delete Vertex`.

To create an edge between two vertices the user should click on one vertex and drag to the vertex that is supposed to be its child. The Millstream system structure is enforced, which forbids any edge between two joints or two nodes. To add links between joints the user holds shift while creating the edge.

When edges are created the layout will be updated, so the vertices position themselves in appropriate positions. To disable this behaviors check the `Lock` check-box below the graph.

To decide which vertices and edges/links that should be on the left-hand side of a rule select them by holding shift and alt while clicking on them. Click the button `Add to LHS` for adding, and `Remove from LHS` for removing. Edges and links between two LHS vertices are automatically added to LHS as well. *Note: Sometimes this functionality stops working, then just click on another rule and reselect the rule.*

By pressing the + and - buttons the graph will be zoomed accordingly. By pressing `Deselect` click-and-drag on any empty space will move the viewing area.

Startgraphs are created in the same way, except that they are not associated with a word and not have the LHS buttons.

### A.4 Modifying a MillSystem

A MillSystem can be opened by selecting `Open MillSystem...` in the File-menu. Be cautious when deleting modules or symbols, since unexpected behavior can occur if they are used in a symbol, rule or startgraph. You can be asked for save the system even if you have not modified it.

### A.5 Running a derivation

When a MillSystem is defined or opened, it is possible to run a derivation. Go to the `Derivation` tab. From the beginning a horizontal line of root vertices are present in the `derivation forest`. These represents the defined startgraphs of the MillSystem. Select the word text-box in the lower left corner and type a word. When pressing enter (or clicking on the `Apply word` button) MillSim will try to apply rules matching that word. In the status-bar it will report how many rules that was associated with that word, and in that case how many of the rules that was matched according to its left-hand side. If at least one rule was matched the derivation forest will be extended with that/those configuration(s). The user can click on any vertex in the derivation forest to see that configuration to the right.

To start over with a derivation click the button `Reset derivation`. 
A.6 Saving a MillSystem

By selecting Save MillSystem... in the File-menu, or try to close MillSim or open another MillSystem, you will be asked to save the system. It should preferably have the extension .xml. Note: The saved system does not include any derivation; only modules, symbols, rules and startgraphs.

A.7 Exporting graphs or rules

All graphs shown visually in MillSim can be exported. By clicking on the Export button below a graph, an export dialog will occur. Normally the formats will crop the picture according to how it looks in the window, so it can be an idea to zoom, move and change window size before exporting. The \texttt{tikz figure in \LaTeX} option will export the whole graph/rule to a \LaTeX document. For some formats there exist options, available from the \texttt{Options...} button.

The default location for saved files is in the users directory. To change that or the filename enter another path or select by pressing the button \textit{Browse}....

A.8 Shortcuts

There exist several shortcuts in MillSim and in its tabs.

- \texttt{Ctrl+N} : Create a new MillSystem
- \texttt{Ctrl+O} : Open a MillSystem
- \texttt{Ctrl+S} : Open a MillSystem
- \texttt{Ctrl+W} : Close MillSim
- \texttt{Ctrl+Tab} : Switch to next tab
- \texttt{Ctrl+Shift+Tab} : Switch to previous tab
- In Modules \texttt{Ctrl+L} : Focus on new module name
- In Rules \texttt{Ctrl+L} : Focus on new rule name
- In Derivation \texttt{Ctrl+L} : Focus on word text-box, and \texttt{Enter} to apply word

A.9 Known bugs and problems

Problem: When creating rules or startgraphs, sometimes all vertices moves to the upper-left corner of the area
Solution: Select another rule (or startgraph) and return again. This is an unresolved bug in JUNG.

Problem: The graph visualization does not update (in rules, startgraphs or derivations).
Solution: Change tab and go back. If it does not work, restart MillSim.

Problem: I cannot add/remove vertices to the LHS when creating/modifying rules.
**Solution:** Select another rule, and then return again. Sometimes it helps to add a few neighboring vertices a time.

**Problem:** I created a MillSystem but ordered the modules in the wrong way.
**Solution:** Open the XML file in a text editor and change the order of the lines that looks like `<module>First-module</module>`

**Problem:** Sometimes the children are sorted incorrectly.
**Solution:** Remove the edges to the children and create them in the desired order.

**Problem:** I cannot load my XML file.
**Solution:** Run MillSim from a terminal, and look at the output when trying to open the file, it will probably give you a hint for fixing the XML file manually. If no/insufficient output, change the level of `log4j.logger.millsim.io` to a lower level in the logconfig file. See logconfig for more info. For advanced users, see Appendix B for the XSD schema.
B XSD schema

Listing 1: The XSD schema for Millstream systems in MillSim

```xml
<xml version="1.0" encoding="ISO-8859-1" >
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"

elementFormDefault="qualified">
<! -- Body structure -->
<xs:element name="millsystem">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="modules" type="moduleType" />
      <xs:element name="words" type="wordType" />
      <xs:element name="symbols" type="symbolType" />
      <xs:element name="startgraphs" type="startgraphsType" />
      <xs:element name="rules" type="rulesType" />
    </xs:sequence>
  </xs:complexType>
</xs:element>

<! -- String types -->
<xs:simpleType name="wordStringType">
  <xs:restriction base="xs:string">
    <xs:pattern value="([a-z\-Z\-0-9]+)" />
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="moduleStringType">
  <xs:restriction base="xs:string">
    <xs:pattern value="([a-zA-Z-0-9\-])" />
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="labelStringType">
  <xs:restriction base="xs:string">
    <xs:whiteSpace value="collapse" />
    <xs:pattern value="([a-zA-Z-0-9_.\-])" />
  </xs:restriction>
</xs:simpleType>

<! -- Modules -->
<xs:complexType name="moduleType">
  <xs:sequence minOccurs="0" maxOccurs="unbounded">
    <xs:element name="module" type="moduleStringType" />
  </xs:sequence>
</xs:complexType>

<! -- Words -->
<xs:complexType name="wordType">
  <xs:sequence minOccurs="0" maxOccurs="unbounded">
    <xs:element name="word" type="wordStringType" />
  </xs:sequence>
</xs:complexType>

<! -- Symbols -->
<xs:complexType name="symbolType">
  <xs:sequence minOccurs="0" maxOccurs="unbounded">
    <xs:element name="symbol" type="symbolStringType" />
  </xs:sequence>
</xs:complexType>

<! -- Startgraphs (and general graphs) -->
<xs:complexType name="startgraphsType">
  <xs:sequence>
    <xs:element name="module" type="moduleStringType" />
    <xs:element name="label" type="labelStringType" />
    <xs:element name="rank" type="xs:nonNegativeInteger" />
  </xs:sequence>
</xs:complexType>
```


<xs:element name="startgraph" type="startgraphType" />
</xs:sequence>
</xs:complexType>

<xs:complexType name="startgraphType">
<xs:sequence minOccurs="0" maxOccurs="unbounded">
<xs:element name="module" type="moduleIdType" />
<xs:element name="links" type="linksType" />
</xs:sequence>
</xs:complexType>

<xs:complexType name="moduleIdType">
<xs:sequence minOccurs="0" maxOccurs="unbounded">
<xs:element name="rule" type="ruleType"/>
<xs:unique name="moduleId">
<xs:selector xpath="/module"/>
<xs:field xpath="#module"/>
</xs:unique>
</xs:sequence>
</xs:complexType>

<xs:complexType name="ruleType">
<xs:sequence>
<xs:element name="word" type="wordStringType" />
<xs:sequence minOccurs="0" maxOccurs="unbounded">
<xs:element name="module" type="moduleType"/>
<xs:element name="links" type="linksType"/>
</xs:sequence>
</xs:sequence>
</xs:complexType>
<xs:schema targetNamespace="ruleModuleType"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:complexType name="ruleModuleType">
    <xs:sequence minOccurs="0" maxOccurs="unbounded">
      <xs:element name="module" type="moduleNameType"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="moduleNameType">
    <xs:sequence minOccurs="0" maxOccurs="unbounded">
      <xs:element name="module" type="moduleNameType"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="rulejoinType">
    <xs:sequence minOccurs="0" maxOccurs="unbounded">
      <xs:element name="join" type="ruleJoinType"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="ruleJoinType">
    <xs:sequence minOccurs="0" maxOccurs="unbounded">
      <xs:element name="join" type="ruleJoinType"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="rulenodeType">
    <xs:sequence minOccurs="0" maxOccurs="unbounded">
      <xs:element name="node" type="ruleNodeType"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="ruleNodeType">
    <xs:sequence minOccurs="0" maxOccurs="unbounded">
      <xs:element name="node" type="ruleNodeType"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="rulelinksType">
    <xs:sequence minOccurs="0" maxOccurs="unbounded">
      <xs:element name="link" type="rulelinkType"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="rulelinkType">
    <xs:sequence minOccurs="0" maxOccurs="unbounded">
      <xs:element name="link" type="rulelinkType"/>
    </xs:sequence>
  </xs:complexType>

</xs:schema>