Development of a GXL-GRAIL Serializer/Deserializer

Markus Lindemann
Development of a GXL-GRAIL Serializer/Deserializer

Markus Lindemann
October 2008

Supervisors:
Prof. Dr. Welf Löwe
Phil. Lic. Rüdiger Lincke
Abstract

GRAIL is a Java library for capturing and manipulating graphs. It is used in the VizzAnalyzer reengineering tool developed at Växjö University that allows quality analysis of software systems.

GXL is a standard exchange format for software data in graph structure, mainly used within the field of software reengineering that is widely supported in other tools within the same field. It is important for VizzAnalyzer to support GXL as an exchange format to allow collaboration with other tools on this basis.

As the goal of this thesis, a GXL graph serializer/deserialize architecture for GRAIL has been developed that allows data exchange between VizzAnalyzer and other tools that support the GXL format.

VizzAnalyzer is capable of analyzing large software systems and therefore the task required special attention on high performance and low memory footprint even with large GXL graph structures.
# Table of Contents

Abstract.................................................................................................................................................i

Lists of Figures, Tables and Code........................................................................................................iv

1. Introduction........................................................................................................................................1
   1.1 Problem.........................................................................................................................................2
   1.2 Goals and Criteria.........................................................................................................................2
   1.3 Motivation.....................................................................................................................................3
   1.4 Outline........................................................................................................................................4

2. Background..........................................................................................................................................5
   2.1 VizzAnalyzer...............................................................................................................................5
   2.2 GRAIL...........................................................................................................................................6
   2.3 GXL...............................................................................................................................................6
      2.3.1 Introduction............................................................................................................................6
      2.3.2 Purpose...................................................................................................................................7
      2.3.3 Definition and Features..........................................................................................................8
   2.4 XML...............................................................................................................................................9
      2.4.1 Elements................................................................................................................................10
      2.4.2 Attributes..............................................................................................................................10
      2.4.3 DTD.......................................................................................................................................10
      2.4.4 Well Formed Documents.......................................................................................................10
      2.4.5 Valid Documents....................................................................................................................10
   2.5 SAX...............................................................................................................................................10
   2.6 XML Parsing Approaches Discussion.........................................................................................11
      2.6.1 Introduction............................................................................................................................11
      2.6.2 From Scratch Development..................................................................................................11
      2.6.3 DOM XML-Parsing Framework.............................................................................................12
      2.6.4 Usage of the SAX XML-Parsing Framework.........................................................................12
      2.6.5 Rating and Selection...............................................................................................................12

3. Requirements......................................................................................................................................14
   3.1 Introduction...................................................................................................................................14
   3.2 Features.......................................................................................................................................14
   3.3 Use Cases....................................................................................................................................15
   3.4 Functional Requirements............................................................................................................17
   3.5 Non-functional Requirements.....................................................................................................20
4. Design and Implementation

4.1 Outline of the Solution

4.2 General Design

4.2.1 XML Parsing Framework

4.2.2 VizzAnalyzer Framework Converter

4.3 Components

4.3.1 GXL

4.3.2 SAXGXLReader

4.3.3 SAXGXLWriter

4.3.4 SAXGXLSchemaWriter

4.4 Tests

4.4.1 Testing Approach

4.4.2 Standard Compliance and Content Test

4.4.3 Memory Consumption Test

4.4.4 Processing Speed Test

4.5 Summary

5. Conclusion and Future Work

5.1 Conclusion

5.2 Future Work

5.2.1 Support for Standardized Schemas

5.2.2 Converter Interface Extension

5.2.3 Unified Converter Exception and Warning Handling

References

Appendix A Memory Consumption Test Results

A.1 GXL Writing Memory Overhead

A.2 GXL Reading Memory Overhead

Appendix B Processing Speed Test Results

B.1 Trimmed Mean Calculation

B.2 GXL Writing Speed Measurements

B.3 GXL Reading Speed Measurements

Appendix C Developer Usage Instructions

C.1 Integration of GXL Converter Source Code

C.2 Running GXL Converter Unit Tests

C.3 Adjusting DTD, Schema and Metaschema File References
Lists of Figures, Tables and Code

List of Figures

FIGURE 2.2: Genealogy of GXL (Holt et al. 2002) ..........................................................................................7
FIGURE 3.1: Use case diagram (Panas, Lincke & Löwe 2005, p.17) .................................................................15
FIGURE 3.2: System use case for GXL .............................................................................................................16
FIGURE 4.1: Thesis solution positioning .........................................................................................................21
FIGURE 4.2: VIZZAnalyzer converter integration .............................................................................................22
FIGURE 4.3: SAXGXLReader document parsing simplified sequence diagram .............................................24
FIGURE 4.4: SAXGXLWriter element writing simplified sequence diagram ..................................................26
FIGURE 4.5: GXL converter memory consumption overhead ..........................................................................31
FIGURE 4.6: GXL converter processing speed ................................................................................................33

List of Tables

TABLE 2.1: Requirements for graph-based exchange formats (Holt et al. 2006, p.154) .................................8
TABLE 2.2: XML parsing approaches comparison ..........................................................................................13
TABLE 4.1: GXL attribute type mapping .......................................................................................................25
TABLE 4.2: GXL graph element mapping .......................................................................................................25
TABLE 4.3: GXL graph elements special mappings ........................................................................................26
TABLE 4.4: Addressed requirements and use cases ..........................................................................................34
TABLE A.1: GXL writing memory overhead ..................................................................................................39
TABLE A.2: GXL reading memory overhead ..................................................................................................40
TABLE B.1: GXL writing measured times (Part 1) .........................................................................................42
TABLE B.2: GXL writing measured times (Part 2) .........................................................................................43
TABLE B.3: GXL reading measured times (Part 1) .........................................................................................44
TABLE B.4: GXL reading measured times (Part 2) .........................................................................................45

List of Code

CODE 4.1: GXL node example .........................................................................................................................23
CODE 4.2: GXL graph schema type indicator elements ..................................................................................27
CODE 4.3: GXL schema definition: Graph contains nodes .............................................................................28
CODE 4.4: GXL schema definition: Nodes have an integer property ............................................................29
CODE C.1: Default GXL DTD reference ..........................................................................................................47
CODE C.2: Default GXL metaschema reference .............................................................................................47
CODE C.3: Enabling Local DTD and Schema References ..........................................................................47
1. Introduction

Software development has matured to a point where it goes far beyond programming algorithms and is usually structured within a standardized software development process. The actual production of the software is referred to as software engineering and is dominated by the discussion of fundamental principles, architectural discussions, design approaches and techniques.

Fundamentals like the employed programming paradigm can be discussed on an abstract level whereas comparison of techniques foster more concrete discussions about advantages and disadvantages. These discussions aim towards finding the most ideal approach to produce software.

Friedrich Bauer stated already in 1973 that software engineering is about “the establishment and use of sound engineering principles” (Bauer 1973, p.524). A vast amount of principles exists today with new approaches spawning permanently, which brings forward the inherent question how “sound” these are. It is a crucial question since the choice of principles is one of the determining factors regarding the overall quality of software.

The research approach of software quality analysis has become a field of its own within the broader context of software reengineering. Quality analysis can be regarded as highly relevant due to its ability to provide answers regarding the “soundness” of principles, which is according to Bauer one of the fundamental challenges for software engineering.

Many research tools have been developed so far to support quality analysis of software and researchers work with different concepts and approaches. Collaboration and knowledge transfer are important to the research process in general and are of huge interest regarding results and intermediate products of software reengineering and quality analysis.

The collaboration interest specifically encompasses the need to exchange information gathered with the aforementioned research tools. Information is present within each tool and developers of research tools strive towards exchanging that information across application borders to support consecutive distributed processing. Information exchange is only possible with a common language and when it comes to software tools, there is the more specific need of a common format that is understood by the involved tools. The topic of this thesis involves one specific information exchange language: The Graph eXchange Language (GXL), that aims towards exchanging data between reengineering tools.

One of those tools is the VizzAnalyzer framework initially developed at Växjö University, that uses the GRAIL Java library for internal information representation. It is a tool for quality analysis and focuses on measurements, metrics and visualization. Växjö university collaborates with other groups in the field of software reengineering and quality analysis and industrial partners. These collaboration activities motivate the need to strengthen interoperability of the VizzAnalyzer framework with other tools in the same field.

The topic of this thesis is the extension of the VizzAnalyzer tool with a converter between GRAIL graphs and the established and widely supported information exchange format GXL. The result should strengthen the position of VizzAnalyzer among other software reengineering tools due to enhanced collaboration capabilities.
1.1 Problem

The context of this thesis is the VizzAnalyzer framework developed at Växjö University. The framework is capable of source code information extraction, program information analysis and result visualization.

The starting point for this thesis is the following problem background:

*Grail is a Java library for capturing and manipulating graphs. Our own tools like the VizzAnalyzer and vizz3d use Grail for representing their internal structures. GXL (Graph eXchange Language) is an XML based serialization format for graphs and relations. It aims at serving as a general exchange format for tools handling and manipulating graphs and relations. In order to connect our tools to others, we need an adapter, serializing and deserializing Grail graphs to and from GXL.*

Thus, the problem addressed by this thesis is:

*The practical goal of the thesis project is design a GXL serializer/deserializer architecture for Grail and implement it in Java.*

VizzAnalyzer already supports different means of information externalization like for example the GML file format. However, the current limitation of information import and export capabilities still raises a problem which isolates VizzAnalyzer to a certain degree from other software tools and weakens its applicability. Implementing a GXL-GRAIL serializer/deserializer should solve the import/export needs of the VizzAnalyzer regarding the GXL format.

This task is difficult to achieve: Both GRAIL and GXL are complex and advanced techniques for representing graph structures. It is necessary to investigate and design content mapping between the two different approaches for both transformation directions. GXL introduces the additional challenge of individual schemas as a complement to each individual graph.

Several different implementation strategies appear possible and it is crucial for a successful solution to identify suitable approaches. Furthermore, various additional goals and criteria that apply to the problem have to be taken into account, which are introduced in the following chapter.

1.2 Goals and Criteria

The abstract motivational goal is to enhance information exchange and therefore interoperability of the VizzAnalyzer framework. As stated earlier in subchapter 1.1, the practical goal of this thesis is the design and implementation of a GRAIL - GXL serializer/deserializer for the VizzAnalyzer framework.

In the following, goals are listed and an accomplishment criterion is assigned to each of them:

- **GXL serialization:** The implementation is supposed to allow serialization of runtime data into the GXL format. As mentioned earlier in the problem description, the VizzAnalyzer framework uses the Java library GRAIL to represent the data structure internally. The serialization part therefore has to accept a GRAIL representation of a data structure, be able to interpret the contents
and produce a representation of it in GXL compatible format. The criterion for this goal is, that the implementation can serialize a given GRAIL graph into GXL format, which can be successfully validated.

- **GXL deserialization**: The implementation is supposed to allow deserialization of a graph in GXL format and create an in-memory representation of it, using the GRAIL library. The criterion for this goal is that the implementation can deserialize a graph in valid GXL format, producing a GRAIL graph that can be used internally in the VizzAnalyzer framework.

- **Support for large data collections**: VizzAnalyzer is capable of handling voluminous graphs utilizing the GRAIL library. The GXL serialization/deserialization solution should not impose drawbacks regarding the size of processable data structures. The criterion for this goal is that the design of the implementation suggests no limitations and it can also be evaluated by example with the help of huge graph structures.

- **Optimized performance**: The aforementioned large data collections should not only be possible to process, it is also necessary to perform the task with maximized processing speed. Since the serialization and deserialization of GXL data structures are tasks that are performed on specific request of the end user, it is crucial to strive towards a high processing speed as a development goal. As a criterion for this goal, the design should suggest performance optimized processing and the success can be evaluated with the help of huge graph structures.

- **Easy maintenance**: The VizzAnalyzer framework development is a group effort and also involves students, which contribute with different components like it is the case with this thesis. Maintainability is therefore relying on a implementation that is easy to understand by developers that are new to the project. Therefore, they are not familiar with the code nor supposed to work on the project for an extended time period. The author of this thesis will not be available for maintenance tasks in the future and is therefore bound to deliver a solution that is easy to comprehend for future change requests. The criterion for this goal is that the implementation, as mentioned in the introduction, follows sound engineering principles. In addition, it should should have features that support the claim and naturally it should provide a sufficient documentation of the source code and external documentation to provide an easy access to the code base.

The goals are the starting point for the requirement analysis in Chapter 3. The criteria are refined and the conclusion in the end documents fulfillment of the goals. New possible goals in VizzAnalyzer development, which are not part of this thesis, might arise during development and are candidates for future work suggested in the end.

### 1.3 Motivation

Even though VizzAnalyzer as a stand-alone tool is not depending on external information input other than source code, there is strong motivation to enhance collaboration capability. VizzAnalyzer is first of all not capable of fulfilling all needs of software analysis and second, enhanced data exchange possibilities might boost popularity of VizzAnalyzer, which is crucial to increase external feedback and as a reason to further development.

GXL is a well established data exchange language for reengineering tools with a broad support among other projects in the research area. It is developed with the fact in
mind that reengineering usually is conducted by utilizing different tools and tries to embrace all data exchange needs imposed by these.

Enhancing VizzAnalyzer with the capability to support the GXL format is a promising venture to strengthen its applicability and its position in the software reengineering community. It allows the usage of VizzAnalyzer within a common reengineering workbench of other well established tools in any step of the reengineering process. Since VizzAnalyzer itself provides support for all steps, it can act as valuable resource to other tools that are more limited in functionality and profit from using it for example to conduct advanced analysis such as architecture recovery or visualization with the integrated Vizz3D framework (Panas, Lincke & Löwe 2005, p.4).

1.4 Outline

The remainder of the thesis is structured as following.

Chapter 2 is intended as a reference to provide background information to the reader about different topics that are of interest in the following chapters.

Chapter 3 defines requirements for a successful solution to the problem.

Chapter 4 describes the developed solution from a conceptual level down to selected details of interest in the concrete implementation.

Chapter 5 evaluates and documents the achievement of the objectives. A prospect on possible future work concludes the thesis.

The Appendix contains additional documentation such as detailed test results and developer usage instructions.
2. Background

This chapter provides background information about different topics that are of interest in the following chapters. In particular, the VizzAnalyzer Software and the included GRAIL library is introduced, to which the implementation belongs that is part of this thesis. Additionally, different relevant XML technology background topics are explained.

2.1 VizzAnalyzer

VizzAnalyzer is software reengineering tool. It is based on the extensible VizzAnalyzer framework that allows rapid composition of reengineering tools. The VizzAnalyzer tool is targeted at software quality analysis and can be used as a stand-alone solution. The foundation for analysis is the software source code, where information is extracted into VizzAnalyzer. Different analysis components can be used to process the information and derive additional information. The extracted and derived information results can be visualized with a variety of visualization tools. These three main function areas - “Retrieval”, “Analysis” and “Visualization” - are referred to as “Hot-Spots” within the VizzAnalyzer.

![VizzAnalyzer Framework Architecture](Panas, Lincke & Löwe 2005, p.14)

Figure 2.1: VizzAnalyzer Framework Architecture (Panas, Lincke & Löwe 2005, p.14)
The information that is subject to analysis can also be converted from and to VizzAnalyzer with the help of different converters that each support a specific kind of external representation. One kind of external representation are file formats, that allow serialization of the in-memory representation. A converter for VizzAnalyzer that supports serialization into files following the GXL standard was implemented as part of this thesis.

2.2 GRAIL

VizzAnalyzer requires random in-memory access to the software source code information derived from the actual source code. GRAIL is the Java graph library that is used within the VizzAnalyzer for internal data representation. It consists of interfaces and classes that are instantiated to represent the information in a graph structure with node and edge instances as main artifacts. Attributes are attached as key and value object instances. A GRAIL graph instance is the entry point to access the contained nodes and edges at runtime. A VizzAnalyzer converter is responsible to convert between a GRAIL graph instance with respective node and edge instances and the external representation supported by the converter.

2.3 GXL

2.3.1 Introduction

GXL stands for Graph eXchange Language. The main purpose is to provide a standard XML based exchange format for software data in a graph structure (Holt, Winter & Schurr 2000, p.1). The format is used mainly within the field of software reengineering. After initial discussions in 1998, the development advanced during conferences and workshops mainly in the year 2000 towards an initial prototype of GXL, that was presented at the ICSE 2000 Workshop.

GXL attempts to be a general graph exchange format for the software reengineering community and consequently the initial prototype resulted from a merger between three preceeding graph interchange standards, the “GRAph eXchange format” (GraX), the Tuple Attribute Language (TA) and the file format from the PROGRES graph rewriting system (Holt et al. 2006, p.155). Further discussions led to version 1.0 of GXL, which was ratified as a standard exchange format in software reengineering at the Dagstuhl Seminar “Interoperability of Reengineering Tools” in January 2001.
As the genealogy of GXL shows, the GXL standardization process encompassed presentations and discussions on several occasions, which produced a couple of refined versions prior to the ratified version 1.0 standard. No new major releases were made since then and a wide array of software engineering tools have adopted the standard. Further development of a “Graph Transformation Language” (GTXL) builds upon it as a fundament.

### 2.3.2 Purpose

The main purpose of the GXL standard is *information exchange*. Within the field of software reengineering a huge amount of software tools are already available. The authors of GXL refer to combinations of these tools used by researchers as a reengineering workbench and identify three different types of tools: *Extractors* that collect information from software artefacts, *Abstractors* for changing the form of the information and creating further information by performing analysis and *Visualizers* that display the information in various forms (Holt et al. 2006, p.151).

These tools rely on different internal information representation and combined usage can only occur, if there exists a possibility of information exchange, which is possible for example with a converter between individual information formats. Converters of this kind already exist, but that approach requires a converter for every pair of information format. GXL however strives to be a standard exchange format for all kinds of tools, acting as an intermediary format that supports all kind of requirements.

The developers of the GXL format claim that they have succeeded with the development of such a widely acceptable intermediary format and base that claim on the GXL feature set discussed in the next chapter.
2.3.3 Definition and Features

The foundation for the GXL feature set are various requirements. These requirements were determined and refined by the developers and members of the software reengineering community, which were involved in the process during several workshops and conferences.

In 2001, Winter (Andreas Winter 2001, p.9) presented the following general requirements for exchange formats:

- Independence from specific reengineering dimensions, applications and tools
- Concrete enough to be interpreted by different reengineering tools

Since the internal representation of information within a variety of reengineering tools is based on graphs, a common ground to fulfill these general requirements lead to a graph structure of attributed, typed and directed graphs.

The analysis of several formats used within a selection of software reengineering tools provided identification of requirements and GXL addresses those with a corresponding set of features:

<table>
<thead>
<tr>
<th>GXL Features</th>
<th>Requirements for Exchange Formats</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Universality</td>
</tr>
<tr>
<td>Graph elements</td>
<td>●</td>
</tr>
<tr>
<td>Hyperedges</td>
<td>●</td>
</tr>
<tr>
<td>First class Elements</td>
<td>●</td>
</tr>
<tr>
<td>Attributes</td>
<td>●</td>
</tr>
<tr>
<td>Ordering</td>
<td>●</td>
</tr>
<tr>
<td>Hierarchy</td>
<td>●</td>
</tr>
<tr>
<td>Graph schemas</td>
<td></td>
</tr>
<tr>
<td>Extension points</td>
<td></td>
</tr>
<tr>
<td>Simplicity</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: Requirements for graph-based exchange formats (Holt et al. 2006, p.154)

The GXL Features are explained by Holt et al. as follows (Holt et al. 2006, p.153):

- **Graph elements**: Basic graph elements like nodes, directed and undirected edges and attributes must be supported. For maximal flexibility, we permit both directed and undirected edges in the same graph.
- **Hyperedges**: N-ary relationships (hyperedges) must be supported natively. Tools or formats that use hyperedges need to be able to use the exchange format as well. Mapping n-ary relationships onto special nodes and binary edges is an unsatisfactory work-around that does not provide equivalent structural characteristics.
- **First class elements**: Nodes, edges, and hyperedges must be identifiable first-class elements, or objects, such that they can have unique identifiers. Viewing edges as first class
elements treats them as equal to nodes and enables multiple edges between nodes.

- **Attributes**: All graph elements may have attributes added to them. This also includes the attributes themselves, e.g. to express layout features of attributes.
- **Ordering**: Ordering of incidences, i.e. the order of edges incident on a node, must be available such that ordered lists of parameters or declarations can be conveniently expressed.
- **Hierarchy**: Hierarchical graphs must be supported to provide simple sub-structuring of graphs. Subgraphs may be exchanged as separate documents.
- **Graph schemas**: The format must be able to define graph classes, or schemas. These are needed to constrain the form of graphs used in different domains of application. These graph schemas permit the specification and use of types.
- **Extension points**: The exchange language syntax has to be extensible, so that the format can be easily adapted to other areas. Furthermore, extension points must be available to permit enhancement of the language.
- **Simplicity**: The exchange format has to be simple, so it can be read and understood by humans. This feature is achieved through a document type definition with a modest number of elements and corresponding exchange documents that are also small.

The GXL exchange format is based on XML documents and it is defined in a single document type definition (DTD) (Andy Schurr et al. 2002). The DTD enables the creation of conforming document instances, which are structured according to the GXL First Directive (Andreas Winter 2001, p.9):

> Everything is a typed attributed directed graph

Even though the data serialization of both graph and schema information are enabled by this DTD, the creation of standardized schemas is a task within the field of semantics. Therefore, such are not part of the current version of the GXL standard, which defines a common syntax to foster interoperability (Holt et al. 2006, p.156). However, GXL supports the definition of schemas in general.

An important extra feature to ensure that the goal of interoperability can be achieved by GXL are the legal usage conditions. GXL is an open standard without any licensing requirement and just the reproduction of the specification has to be accompanied by an explicit acknowledgment (Sim & Winter 2001).

### 2.4 XML

XML stands for “Extensible Markup Language” and is an open standard for structured data representation. The GXL standard uses XML as a basis. XML documents consist of text data and XML markup, which structures the inherent information. Unlike for example HTML, the markup is given as a predefined set of keywords. It is possible to freely define and use markup tags as necessary. XML just provides the standard how this markup can be defined and used. Since XML documents are standard human
readable text, they are well suited for information exchange between different computer systems. GXL uses XML to define markup which can be used to describe graph data.

XML documents consist of a header that gives initial information about the document and the actual information that is enclosed in XML elements. Without going into details how an XML document actually looks like, which are provided in a huge amount of XML literature (Harold & Means 2001) (McLaughlin & Edelson 2006), the basic concepts of interest are as follows:

2.4.1 Elements

All data within an XML document is encapsulated in elements, starting with the singular root element. The remaining elements are nested in the root element and elements can be nested in other elements, which leads to a tree structure of an XML document. The GXL standard defines which elements are allowed as part of a GXL document and how they are nested. It is necessary to define a mapping between the GRAIL representation of a graph and the XML elements of a GXL document.

2.4.2 Attributes

Each element in XML can have one or more attributes, which is essentially a named value. It is defined in the GXL standard, which attributes are supposed to be used. However, GXL defines that graph data is mainly stored as textual data content of elements rather than as XML attributes. These elements represent the type of the encapsulated value and therefore the GXL approach provides support for typing.

2.4.3 DTD

A ”Document Type Definition” or DTD formulates constraints regarding an XML document. It defines the allowed elements, attributes and structure of a document that is supposed to follow a certain standard. Therefore, a DTD provides a schema for a certain standard type of XML document instances. GXL is such a standard and a DTD is provided that defines restrictions that apply to all GXL documents.

2.4.4 Well Formed Documents

An XML document is well formed if it follows basic syntactical and grammatical rules. They define how the markups are written, which characters are allowed in different sections and how different structures are allowed to be nested into each other. As long as these rules are followed, arbitrary information content is allowed. Thus, the flexibility of XML to represent different kinds of information. A generic XML parser is required to be able to read any well formed XML document.

2.4.5 Valid Documents

An XML document is valid if it is well formed and additionally follows restrictions regarding the information content, which are usually provided as a schema like a DTD. These restrictions mainly define allowed elements and the structure of the elements within a document. If a document is valid, depends on the schema that it is checked against.

2.5 SAX

SAX is a “de facto” standard API for reading documents in XML format (SAX official website 2008). It uses a stream-parsing approach to provide access to the elements of the document with a callback pattern. The parsing is conducted event-based following a
push model, meaning that the parser triggers events while traversing the document (Wikipedia 2008).

Different types of event handler interfaces are part of SAX and have to be implemented allowing an application to handle callbacks, that correspond to artifacts within an XML document.

A SAX parser does not explicitly provide information about the structure of an XML document or keeps parsing state by tracking for example which XML element is containing another. This has to be done by the handler if needed. All events of the same type like the beginning or end of an element cause the same corresponding callback, which hands over the data that is relevant to the event.

The document is traversed by a SAX parser exactly one time during parsing and artifacts are considered in a serial fashion without keeping previous contents in-memory. A handler can selectively decide which information to keep without the parser constructing in-memory representations of the information thus using up memory corresponding to the document size. Therefore, the available memory is not limiting the document size that a SAX parser can process.

Even though this is an uncommon approach, a SAX parser can also be used to produce an XML document. This provides again the advantage, that the SAX parser does not build up a representation of the XML document in-memory prior writing. Instead, the SAX parser is able to directly stream the desired XML artifacts to a destination like a file on a hard disk, not being limited in document size by the available memory. However, the parser is again unaware of the desired tree structure of the document and just can be instructed to write singular artifacts like element start, element end or character data. The application using SAX for writing has to ensure that the resulting document is well formed and optionally valid if desired.

### 2.6 XML Parsing Approaches Discussion

#### 2.6.1 Introduction

Different approaches and traditional techniques to parsing XML documents in general and GXL documents in particular were evaluated to determine a solution that fulfills the requirements. Three major approaches to the underlying problem of parsing appeared as potential candidates and are presented in the following with their distinctive advantages and disadvantages. The subsequent rating makes the decision processes prior development transparent and presents the result.

#### 2.6.2 From Scratch Development

The complete parser can be developed from scratch taking the bare text input from the reader. The implementation includes methods for identifying XML elements and attributes starting from the character or token level. The converters already present within VizzAnalyzer for other serialization formats follow this development approach. This is probably the most obvious approach, which can be assessed as follows:

**Advantages**
- No further technology needed
- No additional dependencies
- Full control over functionality

**Disadvantages**
- Implementation of low level details necessary
- Complex maintenance of parsing details
- Need to design own parsing strategy
2.6.3 DOM XML-Parsing Framework

GXL is an XML based format and there are a couple of frameworks available for parsing XML data. One of the most well know approaches is DOM, which provides parsing and allows random access to the elements within a document at runtime through a complete in-memory representation. The following was taken into consideration:

Advantages
- Provides a proven approach to parsing
- Takes care of low level details
- Well documented Java standard
- Different implementations available
- Easy usage due to random access to the complete document content at runtime

Disadvantages
- Runtime memory requirement depending on document size: parses the complete document to an in-memory representation.
- Additional dependencies need to be introduced
- Maintenance requires DOM knowledge

2.6.4 Usage of the SAX XML-Parsing Framework

Another well known approach for XML-parsing is the SAX API. It provides a stream parsing approach, where callbacks are initiated for certain parsing events. The following was taken into consideration:

Advantages
- Provides a proven approach to parsing
- Takes care of low level details
- Well documented Java standard
- Different implementations available
- Memory requirement independent from document size due to stream parsing approach

Disadvantages
- Allows only sequential document access
- Additional dependencies need to be introduced
- Maintenance requires SAX knowledge

2.6.5 Rating and Selection

The three approaches have different advantages and disadvantages, which allowed a rather straightforward selection in a process of elimination, considering the determined requirements. The development from scratch stands out, since it on the one hand is the approach used so far within other VizzAnalyzer converters and on the other hand does not use a framework. The two other options form the group of framework based approaches. When it comes to the task of processing XML within a programming language, DOM and SAX are the major traditional techniques. They are supported directly by many programming languages including Java, which contains a reference implementation.

Avoiding reimplementation of low level parsing details is the major decision factor in favor of framework usage. During design phase, no convincing reasons supported implementing XML parsing from scratch while proven frameworks for that specific task are ready at hand.
Deciding between the two framework based approaches could be done with help of one of the goals outlined in Chapter 1.2. It is necessary that the parser supports large data collections. More concrete features and requirements refining this goal are presented later. These indicate that this goal is likely to conflict with a parser that uses DOM, since the XML document size determines the memory consumption of the parser. The concept of DOM includes the creation of an in-memory representation of the whole document, which can be arbitrary accessed via interfaces. The purpose of the parsing is creation of a GRAIL graph in-memory representation. As a consequence, application of the DOM approach would lead to two complete in-memory representations of the same graph during parsing.

The remaining approach of using a SAX parser avoids implementation of low level parsing, promises a determined low memory consumption and doesn't conflict with any goal.

The following table provides an overview of the considered advantages:

<table>
<thead>
<tr>
<th>Advantages</th>
<th>From Scratch Development</th>
<th>DOM XML-parsing</th>
<th>SAX XML-parsing</th>
<th>Importance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory requirement independent from document size due to stream parsing approach</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>required</td>
</tr>
<tr>
<td>Takes care of low level details</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>desired</td>
</tr>
<tr>
<td>Easy usage due to random access to the complete document content at runtime</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>desired</td>
</tr>
<tr>
<td>Provides a proven approach to parsing</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>optional</td>
</tr>
<tr>
<td>Different implementations available</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>optional</td>
</tr>
<tr>
<td>Well documented Java standard</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>optional</td>
</tr>
<tr>
<td>No further technology needed</td>
<td>✓</td>
<td></td>
<td></td>
<td>optional</td>
</tr>
<tr>
<td>No additional dependencies</td>
<td>✓</td>
<td></td>
<td></td>
<td>optional</td>
</tr>
<tr>
<td>Full control over functionality</td>
<td>✓</td>
<td></td>
<td></td>
<td>optional</td>
</tr>
</tbody>
</table>

Table 2.2: XML parsing approaches comparison

Taking the advantages into account, the usage of a SAX parser became the technique of choice used in the converter components presented in Chapter 4.3.

The tests in Chapter 4.4 and a discussion of the final implementation as part of the conclusion towards the end of this thesis provides more insight into requirement fulfillment.
3. Requirements
This chapter defines requirements for a successful solution to the problem.

3.1 Introduction
The starting point to derive requirements for the GXL converter solution are the goals of
the solution, which are presented in the first chapter. These goals and criteria demand
certain features that are described in the following section.

3.2 Features
The GXL converter demands certain features that are derived from the goals and criteria
of Chapter 1.2. Each feature receives a feature number for later reference and is
described briefly.

Feature F1: GXL file save capability
Description: The converter can receive a GRAIL graph and serialize it to a text file
with the data content of the graph in GXL format.

Feature F2: GXL file load capability
Description: The converter can receive a file containing a graph in GXL format and
deserialize it to a GRAIL graph, representing the contents for further use
in the VizzAnalyzer framework.

Feature F3: Support for large GRAIL graph structures
Description: When a huge GRAIL graph structure is handed to the converter, the
whole GXL file is generated by it without requiring a corresponding huge
amount of memory.

Feature F4: Support for large GXL files
Description: When a huge GXL file is deserialized, the converter itself does not
require a corresponding huge amount of memory.

Feature F5: Short processing time
Description: Save and load requests to the converter are handled as fast as possible.

Feature F6: Implementation allows direct XML data content handling
Description: The implementation uses an XML framework that allows direct access to
XML data artefacts in both reading and writing mode without the need to
read or write actual XML text file content.

Feature F7: GXL Schema writing support
Description: The converter can produce a GXL schema file that targets a specific
serialized graph in GXL format.
3.3 Use Cases

This section describes the use cases where interaction with the converter is possible. This interaction is the basis for the functional requirements documented in the following chapter.

The following use case diagram of the VizzAnalyzer documents that only the actor “User” is interacting with the use cases within the “VizzAnalyzer Application” system at runtime.

![Use case diagram](image)

The use case “retrieve program information” might encompass usage of the converter to retrieve graph data from an external GXL file. However, the VizzAnalyzer Framework's architecture (see Figure 3.1) shows that converters form a hot-spot of their own and are not considered as a retrieval wrapper. It is the VizzAnalyzer framework that interacts with the converters and therefore is the actor within converter use cases.

This is also reflected by the fact that the converter needs no user interface of its own. VizzAnalyzer employs a common GUI Dialog for opening a graph from an external file. The GXL converter is used if the file is of GXL format. A complementary use case within the “VizzAnalyzer Application” system in figure 3.1 would be “save program information” that encompasses usage of the converter to save graph data to an external GXL file, the actual user interface is here also part of the VizzAnalyzer application.
For the converter solution, system use cases apply.

![Diagram](Image)

**Figure 3.2: System use case for GXL**

The use cases are described in the following in detail and references to corresponding features from the previous chapter are given:

**Use Case U1: Convert program information to GXL**

**Features#: 1,3,4,5,7**

**Summary:** The framework works internally with the GRAIL graph library to accomplish analyzing and visualizing of software systems. It uses a converter to externalize the data, represented with the help of GRAIL to a GXL format file and a corresponding GXL schema file.

**Actors:** The VizzAnalyzer framework

**Preconditions:** The VizzAnalyzer program including the GXL converter is running and program information is loaded.

**Trigger:** The VizzAnalyzer framework invokes the GXL converter and directs it to convert a GRAIL graph. The higher level trigger for this use case is, that the user of the VizzAnalyzer program tries to save the program information and chooses to do so in GXL format within the corresponding user interface component.

**Basic Flow:**
1. The framework hands over the GRAIL graph and access to the file to the converter.
2. The converter saves the GXL file.
3. The converter saves the corresponding GXL schema file.

**Alternate Flow:** An error occurred, the framework receives an exception.

**Postconditions:** A text file with the GRAIL graph in GXL format and a text file with a corresponding GXL schema is saved to the file system.
Use Case U2: Convert program information from GXL

Summary: The framework works internally with the GRAIL graph library to accomplish analyzing and visualizing of software systems. It uses a converter to import graph data from within a GXL format file.

Actors: The VizzAnalyzer framework

Preconditions: The VizzAnalyzer program, including the GXL converter, is running and a file with a graph in GXL format is available within the file system.

Trigger: The VizzAnalyzer framework invokes the GXL converter and directs it to convert a GXL file. The higher level trigger for this use case is, that the user of the VizzAnalyzer program tries to load a GXL file within the corresponding user interface component.

Basic Flow:
1. The framework forwards access to the desired file to the converter.
2. The converter returns a GRAIL graph to the framework.

Alternate Flow: An error occurred, the framework receives an exception.

Postconditions: A GRAIL graph representing the contents of the GXL file is loaded in VizzAnalyzer.

3.4 Functional Requirements

The functional requirements in this section are binding for the implementation of the converter solution. They are derived from the use cases and features in the previous chapters and a corresponding reference is given.

Requirement R01: VizzAnalyzer converter implementation compatibility

Use Case#: 1, 2
Feature#: 1, 2

Description: The converter shall obey the specifications for the converter hot-spot within the VizzAnalyzer framework.

Rationale: It is necessary, that the converter is implemented in a way that it fits into the corresponding hot-spot area of the overall architecture of VizzAnalyzer, to be usable within the context of the VizzAnalyzer program.

Fit Criterion: The VizzAnalyzer framework can invoke the converter.
Requirement R02: **GRAIL graph compatibility**

**Use Case#: 1, 2**

**Feature#: 1**

**Description:** The converter shall be capable of processing graphs instantiated with the GRAIL library as it was available from Växjö University in January 2008.

**Rationale:** The GRAIL library is an ongoing development effort. However, the implementation that is part of this thesis, focuses on the revision that was provided from Växjö University in January 2008.

**Fit Criterion:** The converter is able to process GRAIL graphs with the intended revision.

---

Requirement R03: **GXL Standard compliance**

**Use Case#: 1**

**Feature#: 1, 7**

**Description:** The resulting GXL-files shall conform to the GXL standard. The converter, developed as part of this thesis, shall comply to the at writing time most recent GXL-Format Version 1.0 from 2002.

**Rationale:** One of the goals of the GXL standard is interoperability between software and can only be achieved with standard compliance.

**Fit Criterion:** GXL files produced by the converter are standard compliant.

---

Requirement R04: **GRAIL graph property support**

**Use Case#: 1**

**Feature#: 1**

**Description:** All properties of a GRAIL graph should be processed.

**Rationale:** The user expects, that no information loss occurs if a graph is saved into GXL and all information is trustworthy encapsulated within the GXL format. However, it might be acceptable to lose information during serialization, if there is a constraint why a particular piece of information from a GRAIL graph instance cannot or should not be serialized and the user should be informed about the loss.

**Fit Criterion:** Appropriate test graphs can be converted without unintended information loss.
<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
<th>Rationale</th>
<th>Fit Criterion</th>
</tr>
</thead>
<tbody>
<tr>
<td>R05: GXL element support</td>
<td>The converter shall be able to deserialize a GXL graph and can interpret the containing artefacts into corresponding GRAIL graph nodes and edges as meaningful as possible.</td>
<td>In general, any given GXL graph shall be deserialized into a GRAIL graph. However, the GXL format is designed to be as encompassing as possible and covers a wide range of graphs. These graphs might contain details that do not fit as intended into a GRAIL graph object. Therefore, the converter should try to be flexible during parsing and extract as much information as possible from a GXL graph.</td>
<td>The converter is able to completely deserialize a GXL graph into GRAIL that contains matchable artefacts. In particular, one that has been created with the converter itself.</td>
</tr>
<tr>
<td>R06: Large GRAIL graph support</td>
<td>The converter shall be capable of processing especially huge GRAIL graph structures without a significant impact on its memory consumption.</td>
<td>The GRAIL framework is supposed to be capable of handling huge data structures. It is necessary that new additional components like the GXL converter are not introducing bottlenecks.</td>
<td>GRAIL graphs, like the ones provided by Växjö University in GML file format as test data, can be processed with no significant impact on memory consumption of the converter.</td>
</tr>
<tr>
<td>R07: Large GXL file support</td>
<td>The converter shall be capable of processing especially huge GXL graph structures without a significant impact on its memory consumption.</td>
<td>The GRAIL framework is supposed to be capable of handling huge data structures. It is necessary that new additional components like the GXL converter are not introducing bottlenecks.</td>
<td>GRAIL graphs, like the ones provided by Växjö University in GML file format as test data, can be processed as GXL files with no significant impact on memory consumption of the converter.</td>
</tr>
</tbody>
</table>
3.5 Non-functional Requirements

The following requirements are not aimed towards functionality and have no directly correspondence to a use case.

**Requirement R08: Performance**

<table>
<thead>
<tr>
<th>Feature#: 5</th>
</tr>
</thead>
</table>

**Description:** The converter shall be implemented with processing performance in mind. Therefore, the requirement for the implementation of the converter is, that it shall avoid introducing any performance critical programming patterns.

**Rationale:** Processing power is an important performance issue to take into account. This applies both for the architectural design decisions and concrete implementation details.

**Fit Criterion:** The converter is able to perform both serialization and deserialization of large graphs, such as the ones provided by Växjö University in GML file format as test data, in reasonable time on an at writing time current personal computer. The time is reasonable, if those test graphs can be processed in a couple of seconds, not minutes.

**Requirement R09: Maintainability**

<table>
<thead>
<tr>
<th>Feature#: 6</th>
</tr>
</thead>
</table>

**Description:** The converter should be maintenance friendly.

**Rationale:** Since the implementation is to be maintained by a development group with the likelihood of high fluctuation, it shall be developed with easy maintenance in mind.

**Fit Criterion:** Design and implementation show characteristics that are well known to ease maintenance. No characteristics should be present that directly suggest maintenance difficulties.
4. Design and Implementation

This chapter describes the developed solution from a conceptual level down to selected details of interest in the concrete implementation. It furthermore contains details about the implemented and performed tests.

4.1 Outline of the Solution

VizzAnalyzer derives GRAIL graph instances from source code. Converters allow to serialize and deserialize a GRAIL graph instance into an external data format. The solution, that is implemented as part of this thesis, is one of those converters. It can be used alongside other converters, as for example the already existing converter for the GML file format.

The GXL converter for GRAIL consists of three major components: The general converter that provides access for the VizzAnalyzer framework, the GXL serializer that allows transformation of a GRAIL graph instance to a GXL representation and the GXL deserializer that interprets a GXL graph and instantiates a corresponding GRAIL graph instance. The GXL serializer features a subcomponent that is responsible for creating GXL schema files corresponding to serialized GRAIL graphs.

4.2 General Design

4.2.1 XML Parsing Framework

The GXL file format is based on XML, therefore the solution has to both read and write XML in order to process graphs. Different approaches to processing generic XML information are described and rated in Chapter 2.6, which led to the selection of a SAX based framework as a design decision.

Since a SAX parser follows a stream parsing approach without loading an XML document entirely into memory, huge GXL files can be imported into VizzAnalyzer without significant impact on memory consumption of the converter (Requirement R07). The converter further avoids instantiation of new objects, duplicating data fields and keeping those instances referenced, preventing garbage collection. As a result, the physical memory of the user's computer should not restrict the size of the GXL document with the converter being the bottleneck.

Application of a SAX parser avoids implementation of low level parsing details. Manual handling of text artefacts tends to be difficult and unclear to implement. Since the usage of a framework on the other hand provides a proven approach to the problem
and enforces clarity, it eases maintenance (Requirement R09). Producing XML with the help of a framework is also a major support to ensure that resulting GXL documents are standard compliant (Requirement R03), since it handles the XML syntax.

Different framework implementations can be used, and there are mature software packages available. The author assumes, that these will offer optimized performance, and if needed choose the implementation that offers the fastest transformation (Requirement R08). The most apparent choice was the reference implementation from Sun Microsystems.

4.2.2 VizzAnalyzer Framework Converter

The converter has to be integrated into the VizzAnalyzer framework structure to be usable. One part of the VizzAnalyzer framework are the hot-spots, where arbitrary reverse engineering components can be connected. Converters are one of these hot-spots and the VizzAnalyzer Handbook defines their role (Panas, Lincke & Löwe 2005, p.16):

A converter transforms external exchange formats, such as GML, to our own internal data representation (GRAIL), and vice versa. Converters are developed once and kept for reuse. Our framework contains a collection of such converters and allows for easy implementation of new ones.

Therefore, the implementation aims to be usable as a converter for the GXL format within VizzAnalyzer. Three stages of framework usage are defined by the authors: Extension at design time, Composition at compile-time and Application run at run-time. The GXL converter is part of the first stage. The use case diagram of the VizzAnalyzer already presented earlier (Figure 3.1) documents the extension possibility.

Within that diagram, the actor “User” can “retrieve program information”, which would lead to usage of the GXL converter if such is provided in GXL format. A user can choose to load a file within the VizzAnalyzer Graphical User Interface (GUI) and consecutively select a GXL file. This instantiates the class GraphLoadSave, which is responsible for choosing the appropriate converter. The current approach is discussed in more detail in the chapter “Future Work”. The modular hot-spot design allowed to integrate the converter along with the already existing converters like for example the GML file format converter by implementing the Converter interface.

Figure 4.2: VizzAnalyzer converter integration
The GXL class is part of the `grail.converters` package while the other components described in the following section form the package `grail.converts.gxl`.

The converter provides methods for loading a graph in the applying format and returning it as an instance of a `GraphInterface` to a GRAIL graph. It also allows to save a GRAIL graph in GXL format and implements methods that closely resemble the ones that can be found in the current GML converter. The converter fits into the corresponding hot-spot area of the VizzAnalyzer framework and is usable by its components (Requirement R01).

4.3 Components

4.3.1 GXL

The GXL class implements the `Converter` interface and it is supposed to serve two distinctive purposes:

- Loading of a given GXL file into memory as a GRAIL graph (Requirement R01, R07)
- Saving a given GRAIL graph as GXL file (Requirement R01, R06)

Both purposes require different implementation approaches, which are encapsulated in additional classes and the main converter class `GXL` is responsible for suitable instantiation and information forwarding. The intention behind this design is enhanced maintainability of the converter code base which supports easy integration of alternate versions or testing of rewritten converter components (Requirement R09).

The program components of the GXL converter are collected in the `grail.converters.gxl` package within the VizzAnalyzer source code and the implementations, provided as part of this thesis, are explained in the following sections.

4.3.2 SAXGXLReader

The `SAXGXLReader` class uses, as the name implies, the SAX framework to parse and convert the contents of a GXL file (Use Case U2). The SAX parser is not aware of the GXL standard and provides generic access to XML artefacts within the provided document. All artefacts of interest that are provided by the SAX parser in the framework have to result in corresponding GRAIL artefacts. GXL graph elements are represented in GRAIL as class instances and GXL attributes correspond to properties and fields in GRAIL. Every encountered node, edge and attribute from the GXL graph is imported (Requirement R05).

Nodes, edges and their attributes are represented as nested XML elements. For example, a single node consists of several complete elements:

```xml
...<node id="n0">
  ...
  <attr name="IntProperty1">
    <int>123</int>
  </attr>
  ...
</node>...
```

**Code 4.1: GXL node example**

A SAX parser calls the method `startElement` for the `node`, `attr` and `int` element. However, it does not keep state information about the tree structure and it does not
allow access to previous artefacts. Therefore, the SAXGXLReader is responsible for keeping information details until several details together allow instantiation of a matching GRAIL graph element. The following sequence diagram shows the schematics, that allow to read the previous GXL Node example.

![Sequence Diagram](image)

**Figure 4.3: SAXGXLReader document parsing simplified sequence diagram**

The above diagram shows in condensed form, how the SAXGXLReader implementation accomplishes parsing the simple example of a node with an integer attribute. Each time the parser encounters start and end of an XML element or character data, it is up to the reader implementation to react appropriately by collecting information for later usage or instantiate GRAIL artefacts with the collected information.
The previous GXL code example shows that a node consists of three nested elements, which causes the parser to initiate three consecutive callbacks to the `startElement` method implementation. All different element types of interest are recognized by the SAXGXLReader, including node, attr and int elements. If a node element starts, a new GRAIL node is instantiated (sequence method #3). The name of an attr element is saved, and the nested int element causes the reader to prepare for parsing consecutive character data into an integer value. Character data is reported to the reader by a callback to the `characters` method which saves the content (sequence method #7). The parser then encounters three consecutive end tags. The end of the attr element leads to instantiation of GRAIL key and value objects according to the previous saved information and the property is added to the GRAIL node (sequence method #10). The node end tag causes the reader to prepare for a new GRAIL graph element.

The keys in GRAIL properties are supposed to be references to `GraphProperties` class instances. Each instance encapsulates an attribute name combined with the class type, that is acceptable as a value for that name and a corresponding implementation of a string converter. Recurring attribute names will foster reusage of the previously instantiated `GraphProperties` instance. Therefore, each attribute name can only be mapped to one acceptable type and corresponding string converter during runtime.

The SAXGXLReader fetches the name of the GRAIL property from the attr element opening tag and derives the class of acceptable values and the corresponding GRAIL `PropertyStringConverter` from the nested element name.

<table>
<thead>
<tr>
<th>GXL attribute type element name</th>
<th>GRAIL GraphProperties acceptable value class</th>
<th>GRAIL GraphProperties PropertyStringConverter</th>
</tr>
</thead>
<tbody>
<tr>
<td>“bool”</td>
<td>Boolean.class</td>
<td><code>IntegerStringConverter</code></td>
</tr>
<tr>
<td>“int”</td>
<td>Integer.class</td>
<td><code>IntegerStringConverter</code></td>
</tr>
<tr>
<td>“float”</td>
<td>Double.class</td>
<td><code>DoubleStringConverter</code></td>
</tr>
<tr>
<td>“string”</td>
<td>String.class</td>
<td><code>PropertyStringConverter</code></td>
</tr>
</tbody>
</table>

Table 4.1: GXL attribute type mapping

For each property a scope is defined, where it shall be applied. It is indicated by a static integer value set for each instance of the `GraphProperties` class. The following table shows which GRAIL class is instantiated by the SAXGXLReader to represent a certain GXL graph element. It further shows the static integer that defines the scope, which applies to nested GXL attributes.

<table>
<thead>
<tr>
<th>GXL graph element</th>
<th>GRAIL class</th>
<th>GraphProperty kind of GXL graph element attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>graph</td>
<td><code>SetBasedDirectedGraph</code></td>
<td><code>GraphProperties.PROPERTY_ALL</code></td>
</tr>
<tr>
<td>node</td>
<td><code>SetBasedDirectedNode</code></td>
<td><code>GraphProperties.PROPERTY_NODE</code></td>
</tr>
<tr>
<td>edge</td>
<td><code>SetBasedDirectedEdge</code></td>
<td><code>GraphProperties.PROPERTY_EDGE</code></td>
</tr>
</tbody>
</table>

Table 4.2: GXL graph element mapping

Certain GXL graph elements are identified during the parsing process and have special mappings into the GRAIL graph library according to the following table.
GXL Element | Mapped to GRAIL as
---|---
node attribute “id” | protected field key in the corresponding node instance
edge attribute “from” | field with the source node instance kept within the edge instance
edge attribute “to” | field with the target node instance kept within the edge instance

Table 4.3: GXL graph elements special mappings

GXL indicates source and target of a relation using corresponding node identifiers as edge attributes. Since GRAIL relies internally on new instantiated id objects, the concrete identifier value is lost during import. The loss is rated as acceptable and intended, since it is the relation that is of interest, not the specific value of the node id (Requirement 05).

4.3.3 SAXGXLWriter

In order to provide serialization capabilities (Use Case U1), the converter solution encompasses the SAXGXLWriter class. Even though DOM is the common approach to create XML files, the goal of support for large data collections, which is manifested in requirement R07, led to the decision of using SAX. The SAX framework is well known as an XML parsing framework and used as such in the SAXGXLReader, which is also part of the implementation. SAX is used in the SAXGXLWriter to produce the concrete XML document from provided content.

The information source is an instance of a GRAIL graph, that is traversed with library inherent features, and every piece of information is converted to a corresponding XML element (Requirement R02,R04).

![figure](image)

**Figure 4.4: SAXGXLWriter element writing simplified sequence diagram**

The above figure shows how the implementation of the SAXGXLWriter works on a conceptual level. It fetches pieces of information from each GRAIL graph element
while traversing all the elements like nodes, edges and their inherent properties, which have to be written as XML elements and attributes.

To define attributes for an XML element, SAX employs an instance of the `AttributesImpl` class, that has to be populated with the desired name and value pairs for each attribute (sequence method #2). The `SAXGXLWriter` finally instructs the SAX parser to write an XML start tag with the attributes encapsulated in the `AttributesImpl` instance (sequence method #3).

Required attributes within GXL such as the “id” attribute of a “graph” element are always set by providing defaults, if no such information is present within the GRAIL graph to ensure compliance with the GXL standard (Requirement R03).

Relations within a GRAIL graph are represented by runtime object-references to the corresponding endpoint node instances. GXL on the other hand requires the “id” attribute for nodes and edges contain attributes that reference those identifiers to indicate a relation. Node instances in GRAIL have a “key” field, but since it is of type `Object`, it can not be assured that a suitable identifier string can be derived from it. Therefore, the `SAXGXLWriter` generates integer identifiers for the nodes. It produces corresponding reference attributes within edges to completely serialize the information contained in the given GRAIL graph.

The value space for these XML identifier references is defined by the NCName production (World Wide Web Consortium 2001). This production defines that an identifier value has to begin with a letter (World Wide Web Consortium 1999), therefore the character 'n' is added as a prefix to each generated integer id.

Graph elements such as the graph itself and the contained nodes and edges are supposed to be valid according to a GXL schema. A nested “type” element connects each of those GXL graph elements to a type defined in the related GXL schema.

```xml
...<graph id="Grail2GXLGraph">
  <type xlink:href="schemafile.gxl#GRAILGraph" />
  <node id="n0">
    <type xlink:href="schemafile.gxl#GRAILNode" />
  </node>
...```

**Code 4.2: GXL graph schema type indicator elements**

The above example shows a GXL graph that has elements which are instances of types from the “schemafile.gxl” schema. The `SAXGXLWriter` instantiates a `SAXGXLSchemaWriter` prior writing the GXL graph. During graph writing, the schema writer is instructed with necessary information to subsequently produce a valid corresponding GXL schema.

**4.3.4 SAXGXLSchemaWriter**

A `SAXGXLSchemaWriter` is instantiated by a `SAXGXLWriter` before a GRAIL graph is serialized. It is supposed to write a standard compliant GXL schema, that documents GRAIL inherent restrictions, that apply to the GXL graph to be written (Requirement R03). Therefore, different GRAIL graphs will potentially lead to different GXL schemas. If an external application respects the schema during modifying the graph, it should be possible to reimport the altered graph into GRAIL. However, the `SAXGXLReader` is not strict regarding schemas, since it strives to import every GXL graph as meaningful as possible.

The produced schema has to comply to the GXL metaschema, which defines the possible schema artifacts. The schema writer uses SAX to actually write XML content.
The approach to produce XML elements and attributes is described in the SAXGXLWriter chapter.

Some parts of each schema are generic for all graphs written by the SAXGXLWriter, such as that a graph contains nodes and edges. The schema writer documents those containment relationships like the following example for nodes demonstrates:

```
...<node id="GRAILGraph">
    <type xlink:href="gxl-1.0.gxl#GraphClass" />
    <attr name="name">
        <string>GRAILGraph</string>
    </attr>
</node>
...<node id="GRAILNode">
    <type xlink:href="gxl-1.0.gxl#NodeClass" />
    <attr name="name">
        <string>GRAILNode</string>
    </attr>
...<edge id="e0" from="GRAILGraph" to="GRAILNode">
    <type xlink:href="gxl-1.0.gxl#contains" />
</edge>
...
```

**Code 4.3: GXL schema definition: graph contains nodes**

The GXL schema excerpt above demonstrates that every type GRAILGraph in a GXL document, denoted with the matching type attribute (see Code 4.2), is of type GraphClass from the metaschema. The metaschema is here expected in a file named “gxl-1.0.gxl”. Appendix C.3 contains details about handling the metaschema file location. The second schema node assigns the meta type NodeClass to each GRAILNode in a GXL document. Finally, the contains edge documents, that a GRAILGraph contains GRAILNodes.

However, other parts of the schema are specific to the GXL document for a given GRAIL graph. As described earlier, it is a restriction of GraphProperties, that each property name can only correspond to one value type like boolean or integer. The SAXGXSLSchemaWriter features the public HashMap<String, String> that is populated by the SAXGXLWriter with name and type pairs during production of the GXL document. The schema writer uses those mappings to produce according restrictions within the GXL schema document. Consider the example of an integer property named IntProperty1 (see Code 4.1), that causes the SAXGXLWriter to populate the map with a “IntProperty1” to “int” mapping. The schema writer will regard this mapping by producing the following XML structure in the schema document.
The first node defines that an attribute named “IntProperty1” exists. The following edge defines that this attribute has values from the integer domain, meaning that it can only contain integer values. The second edge allows the attribute to be assigned to a node. The final node references the integer type defined in the metaschema. Note that this kind of schema restrictions apply only on a per-graph level, therefore each graph will receive its specific schema.

4.4 Tests

4.4.1 Testing Approach

Different types of tests were conducted on the implementation, which show that relevant requirements are fulfilled. A variety of graphs were provided by Växjö University in GML file format to allow testing of the converter. Within the VizzAnalyzer framework, unit tests are supposed to be developed for assessment purposes.

Different test cases for the GXL converter were implemented in the class `test.grail.TestGXL` that cover serialization and deserialization functionality. The test graphs also included typical large graphs that allowed assessment about memory consumption and processing speed of the converter during runtime. In addition, the helper class `GRAILTestUtils` was implemented. It contains methods to construct generic test graphs with a desired number of nodes, edges and properties to support more detailed analysis about the runtime performance behavior of the converter. Furthermore, the `GRAILTestUtils` encompass a method to print out detailed informations about a given GRAIL graph and its nodes and edges to allow testing of GXL converter import functionality.

The testing areas covered by unit tests are presented in the following subchapters.

4.4.2 Standard Compliance and Content Test

The `SAXGXLWriter` produces GXL documents and the `SAXGXLSchemaWriter` produces matching GXL schema documents. The GXL developers provide a validator (GUPRO 2005), that can be used to test the following details:

- If GXL documents and GXL schema documents are valid XML files according to the GXL DTD.
● If a GXL schema document is valid according to the GXL metaschema.
● If a GXL document is valid according to a GXL schema document.

The GXL documents and schemas resulting from the test graphs were tested with this validator. Since the GXL validator is a command line tool, it was invoked with the GXL document name as an argument and both schema and metaschema needed to be present in the same directory. The results confirmed, that the SAXGXLWriter and the SAXGXLSchemaWriter components produce valid GXL documents (Requirement R03).

These valid files were then converted from GXL back into a GRAIL graph instance. To ensure that the converter is not just parsing a GXL file but also constructing a GRAIL graph with corresponding contained artifacts, the GRAILTestUtils class was used to print out extensive details and compare the initial GRAIL graph to the reimported instance. These comparison tests showed, that all GRAIL graph properties were serialized into the GXL file by the SAXGXLWriter (Requirement R04). They also showed, that the SAXGXLReader is capable of reading GXL standard compliant files and instantiate matching GRAIL graph artifacts (Requirement R05).

4.4.3 Memory Consumption Test

The SAXGXLWriter and SAXGXLReader components of the converter have a GRAIL graph instance as source or target respectively. The memory consumption of this GRAIL graph instance during operation of the converter is therefore inevitable. However, the converter is required to have no additional significant impact on memory consumption that would qualify as introducing an additional bottleneck (Requirement R06, R07).

Fulfillment of this requirement was tested by performing both serialization and deserialization of large graphs such as the ones provided by Växjö University in GML file format as test data.

In addition to the provided graphs from Växjö University, the helper methods from the GRAILTestUtils class were used to generate generic GRAIL test graphs. These graphs can consist of an arbitrary number of nodes and edges with each a set of test properties covering different types. The memory consumption of the converter was examined for a gradient of graph elements up to graphs with 20000 nodes and 20000 edges.

Several methods to optionally allow measurement and recording of the memory usage of the converter were implemented in the GXL class. The method GXL.setMemoryConsumptionTestmode(boolean) allows to enable the memory usage recording option.

The goal was to show that the test graphs can be processed without significant memory consumption of the converter. Since reading and writing of GXL documents are implemented separately, the tests were also done for both use cases separate.

Different unit tests in the class test.grail.TestGXL are implemented to encompass the different use cases and graphs.

The SAXGXLWriter was tested with the following methods:

testGenericGraphWritingMemoryConsumption()
testSpecificGraphWritingMemoryConsumption()

The SAXGXLReader was tested with these methods:

testGenericGraphReadingMemoryConsumption()
testSpecificGraphReadingMemoryConsumption()
The following figure shows the memory consumption overhead of the converter for graphs of different sizes.

![GXL Converter Memory Consumption Overhead](image)

Figure 4.5: GXL converter memory consumption overhead

As the above figure shows, reading of graphs with up to 40000 graph elements result in a constant low memory consumption overhead of the converter. Both the provided test graphs from Växjö University and all the generic test graphs could be parsed into GRAIL with an overhead of just 16576 bytes. Several test executions for the same graph revealed that the overhead sometimes may vary a couple of bytes upward but mostly results in the previously stated amount. The author assumes that these variances have their cause in incomplete garbage collection of the virtual machine.

The writer component shows a linear increase in memory overhead with increasing graph sizes. A memory heap dump was analyzed to research if this linear increase is justified. It showed that the `HashMap`, necessary for storing the generated identifiers assigned to the GXL nodes, is the cause for this linear increasing memory overhead of the `SAXGXLWriter`.

The large test graphs given by Växjö University contain 21077 graph elements (4856 nodes and 16221 edges) and 22797 graph elements (6186 nodes and 16611 edges). These graphs could both be processed with less then 500 kilobytes memory consumption overhead. The largest generic test graph with 40000 graph elements (20000 nodes and 20000 edges) was processed with less than one megabyte memory consumption overhead.

This memory consumption overhead test results of the converter for both reading and writing are not significantly high, considering the RAM size of at writing time current computers. Therefore, the requirements to support large GRAIL graphs and GXL files are fulfilled (Requirements R06, R07).
4.4.4 Processing Speed Test

In difference to the two previous test areas, the processing speed is directly relating to the hardware of the computer where the conversion is performed on. It is required, that reasonable performance can be achieved on an at writing time current personal computer (Requirement R08). Fulfillment of this requirement was tested on the authors computer (Intel Core2 CPU T7200 with 2GHz) by performing both serialization and deserialization of large graphs such as the ones provided by Växjö University in GML file format as test data.

In addition to the provided graphs from Växjö University, the helper methods from the GRAILTestUtils class were used to generate generic GRAIL test graphs. These graphs can consist of an arbitrary number of nodes and edges with each a set of test properties covering different types. The processing speed was examined for a gradient of graph elements up to graphs with 20000 nodes and 20000 edges. To measure the elapsed time during conversion, unit tests where invoked and the converter implementation was enhanced to print out the amount of milliseconds that the whole conversion took up. Testing showed, that the processing times varied between different executions of the same graph serialization or deserialization, therefore the unit tests were modified to automatically re-run the same test for a definable amount of executions and print out the milliseconds that each run takes.

For each test graph, 20 runs were performed and the values were then used to calculate the average runtime after leaving away the outliers, as shown in more detail in appendix B. The goal was to show that those test graphs can be processed in a couple of seconds, not minutes. Since reading and writing of GXL documents are implemented separately, the tests were also done for both use cases separate.

Different unit tests in the class test.grail.TestGXL are implemented to encompass the different use cases and graphs.

The SAXGXLWriter was tested with the following methods:
- testGenericGraphWritingProcessingSpeed()
- testSpecificGraphWritingProcessingSpeed()

The SAXGXLReader was tested with these methods:
- testGenericGraphReadingProcessingSpeed()
- testSpecificGraphReadingProcessingSpeed()
The following figure shows processing durations in milliseconds for graphs of different sizes.

![GXL Converter Processing Speed](image)

**Figure 4.6: GXL converter processing speed**

As the above figure shows, the processing duration for both serialization and deserialization of GRAIL graphs increases linear with the amount of nodes and edges that a given graph has. The large test graphs given by Växjö University contain 21077 graph elements (4856 nodes and 16221 edges) and 22797 graph elements (6186 nodes and 16611 edges). These graphs could both be processed within less then 1.5 seconds which counts as reasonably fast to fulfill the expectations (Requirement R08). The generic test graphs contained each an identical number of nodes and edges and even the largest one used in this test with 40000 graph elements (20000 nodes and 20000 edges) could be processed reasonably fast within approximately 3 seconds.
4.5 Summary

Design and implementation of the GXL converter as a whole and within the separate components were presented in this chapter. The following table summarizes the foregoing details:

<table>
<thead>
<tr>
<th>Design and Implementation Detail</th>
<th>Addressed Requirements and Use Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R01</td>
</tr>
<tr>
<td>XML parsing framework</td>
<td></td>
</tr>
<tr>
<td>GXL class</td>
<td></td>
</tr>
<tr>
<td>SAXGXLWriter</td>
<td></td>
</tr>
<tr>
<td>SAXGXLReader</td>
<td></td>
</tr>
<tr>
<td>GRAILTestUtils</td>
<td></td>
</tr>
<tr>
<td>Tests</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.4: Addressed requirements and use cases

The table outlines, that all functional and nonfunctional requirements and their underlying use cases documented in this thesis are successfully addressed by the developed GXL converter. Moreover, the table gives an overview, which requirement or use case is addressed by each of the concrete parts of design and implementation.
5. Conclusion and Future Work

This chapter will conclude the thesis with a retrospect to the initial problem and present the GXL converter implementation as a solution. In addition, it will point out details that remain as possible future work.

5.1 Conclusion

The abstract motivational goal was to enhance information exchange and therefore interoperability of the VizzAnalyzer framework. It was described and motivated in Chapter 1 how the addition of a GXL converter component can enhance this interoperability.

The starting point for this thesis was the following problem:

The practical goal of the thesis project is design a GXL serializer/deserializer architecture for Grail and implement it in Java.

Goals and criteria were derived, which consequently lead to concrete requirements described in Chapter 3. Chapter 4 described the implementation striving to meet the requirements. Consequently, this implementation should also fulfill the initial goals. In the following, these goals are listed and the accomplishment of the assigned criterion is examined:

- **GXL serialization**: The most obvious goals that were derived from the problem is the ability to serialize and deserialize a graph from within VizzAnalyzer to and from GXL. The implementation addresses the serialization goal with the SAXGXLWriter component. This component accepts a GRAIL representation of a data structure, is able to interpret the contents and produce a representation of it in GXL compatible format, which can be successfully validated against the produced GXL schema.

- **GXL deserialization**: The second obvious goal was, that the implementation is supposed to allow deserialization of a graph in GXL format and create an in-memory representation of it using the GRAIL library. The implementation addresses this goal with the SAXGXLReader. It meets the criterion for this goal by being able to deserialize a graph in valid GXL format and producing a GRAIL graph that can be used internally in the VizzAnalyzer framework.

- **Support for large data collections**: Further investigation into the topic has shown, that memory consumption is an important issues for a VizzAnalyzer converter, which led to the goal that large graphs should be processed with low memory consumption on the researchers workstation. The underlying design of the implementation aims specifically towards this goal. It proved to be successful in tests with very large graphs, that can be serialized and deserialized with a permanent low memory consumption of the GXL converter. The criterion for this goal is met, since the design of the implementation suggests no critical limitations regarding this topic and it could also be evaluated with the help of huge graph structures.

- **Optimized performance**: Processing performance is of high interest for a VizzAnalyzer converter, which led to the goal that large graphs should be processed with maximized processing speed. The design of the implementation addresses this goal and therefore meets the criterion. The success was evaluated in tests with very large graphs that could be processed in a matter of seconds.
- **Easy maintenance:** Finally, the solution was supposed to be maintenance friendly. An important feature regarding this area is the usage of a framework for low level details. This feature combined with a well designed and completely commented code base suggests that this goal is also reached.

The above list summarizes that the initial goals are reached. On top of those, the more concrete features and use cases that apply to the converter were defined. These lead to requirements that were binding for the implementation. Those requirements are met, and consequently the use cases and features are handled by the converter as documented in this thesis.

In conclusion, the initial problem of this thesis is solved.

### 5.2 Future Work

#### 5.2.1 Support for Standardized Schemas

The thesis solution supports creation of schemas. Those schemas are documenting restrictions regarding individual VizzAnalyzer graphs, enabling conflict free reimport of those graphs.

However, the concrete element names have their origin from within GRAIL and their semantical content can not be communicated in GXL schemas. As the authors of GXL point out, it requires external standardization effort (Holt et al. 2006, p.156):

> Since GXL specifies only graphs, it remains to standardize schemas to further describe what these graphs represent.

> In other words, standard schemas, or reference schemas, are needed for being fully interoperable to data interchange.

If standard schemas are available that seem to be appropriate to support, it requires future work to design how that support might be implemented into the GXL converter and the VizzAnalyzer.

#### 5.2.2 Converter Interface Extension

Converters are contained in a **hot-spot** within the VizzAnalyzer architecture. Chapter 4.2.2 describes the integration of the GXL serializer/deserializer as an implementation of the the Converter interface. This interface however just defines two methods `from(String text)` and `from(Reader reader)` that require an implementation to derive a `GraphInterface` instance from a document in text form or from a file containing such document. The interface does not define methods to externalize a GRAIL graph, and the existing converters already indicate the lack by providing inconsistent methods for that task. These methods have differing method names and signatures (e.g. `toGML` and `toTabTable`) which prevent a generic object oriented usage of different converter implementations. The consequences can be examined in the class `GraphLoadSave.java` that connects the VizzAnalyzer framework with the converters in the **hot-spot** area. The code that loads graphs is interacting with the generic converter interface, whereas saving a graph is implemented individually for each converter implementation.

It remains future work to redesign and enhance the converter interface which also leads to necessary refactoring in all individual converters.
5.2.3 Unified Converter Exception and Warning Handling

The current interface for converters just defines that a converter is allowed to throw an `IOException` when instructed to load a graph from a file source. This allows just very limited reporting of parsing problems to VizzAnalyzer. Within the thesis solution it is possible that SAX parsing exceptions occur, which are not desired to be thrown to VizzAnalyzer. However, some parsing problems can be recovered, but might lead for example to omitted artifacts from a source file. It remains future work to design a common exception handling strategy probably comprising custom convert exceptions that can be used by all converters to report different types of problems. Additionally, it might be of interest to allow converters a generic mechanism to report non fatal issues and warnings to VizzAnalyzer.
References


Schurr, A. et al., 2002. GXL (1.0) - Document Type Definition commented. Available at: http://www.gupro.de/GXL/dtd/gxl-1.0.html [Accessed June 21, 2008].


## Appendix A Memory Consumption Test Results

### A.1 GXL Writing Memory Overhead

The following table shows the detailed results of the memory consumption tests for GXL document writing described in Chapter 4.4.3.

<table>
<thead>
<tr>
<th>Graph Elements</th>
<th>Writing Base Memory (Bytes)</th>
<th>Writing Max Memory (Bytes)</th>
<th>Writing Overhead (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>512.328</td>
<td>574.576</td>
<td>62.248</td>
</tr>
<tr>
<td>2000</td>
<td>1.678.248</td>
<td>1.788.840</td>
<td>110.592</td>
</tr>
<tr>
<td>4000</td>
<td>2.880.912</td>
<td>3.006.312</td>
<td>125.400</td>
</tr>
<tr>
<td>6000</td>
<td>4.047.568</td>
<td>4.212.968</td>
<td>165.400</td>
</tr>
<tr>
<td>8000</td>
<td>5.218.816</td>
<td>5.440.600</td>
<td>221.784</td>
</tr>
<tr>
<td>10000</td>
<td>6.397.328</td>
<td>6.658.968</td>
<td>261.640</td>
</tr>
<tr>
<td>12000</td>
<td>7.553.520</td>
<td>7.855.304</td>
<td>301.784</td>
</tr>
<tr>
<td>14000</td>
<td>8.740.912</td>
<td>9.115.816</td>
<td>374.904</td>
</tr>
<tr>
<td>16000</td>
<td>9.895.136</td>
<td>10.310.392</td>
<td>415.256</td>
</tr>
<tr>
<td>18000</td>
<td>11.050.200</td>
<td>11.505.280</td>
<td>455.080</td>
</tr>
<tr>
<td>20000</td>
<td>12.253.640</td>
<td>12.748.544</td>
<td>494.904</td>
</tr>
<tr>
<td>22000</td>
<td>13.408.856</td>
<td>13.943.760</td>
<td>534.904</td>
</tr>
<tr>
<td>22797</td>
<td>15.993.424</td>
<td>16.472.888</td>
<td>479.464</td>
</tr>
<tr>
<td>24000</td>
<td>14.563.768</td>
<td>15.138.672</td>
<td>574.904</td>
</tr>
<tr>
<td>26000</td>
<td>15.783.752</td>
<td>16.463.520</td>
<td>679.768</td>
</tr>
<tr>
<td>28000</td>
<td>16.937.400</td>
<td>17.657.488</td>
<td>720.088</td>
</tr>
<tr>
<td>30000</td>
<td>18.094.008</td>
<td>18.854.272</td>
<td>760.264</td>
</tr>
<tr>
<td>32000</td>
<td>19.247.000</td>
<td>20.047.088</td>
<td>800.088</td>
</tr>
<tr>
<td>36000</td>
<td>21.556.872</td>
<td>22.436.960</td>
<td>880.088</td>
</tr>
<tr>
<td>38000</td>
<td>22.810.728</td>
<td>23.730.816</td>
<td>920.088</td>
</tr>
<tr>
<td>40000</td>
<td>23.964.872</td>
<td>24.924.960</td>
<td>960.088</td>
</tr>
</tbody>
</table>

Table A.1: GXL Writing memory overhead
### A.2 GXL Reading Memory Overhead

The following table shows the detailed results of the memory consumption tests for GXL document reading described in Chapter 4.4.3.

<table>
<thead>
<tr>
<th>Graph Elements</th>
<th>Reading Base Memory (Bytes)</th>
<th>Reading Max Memory (Bytes)</th>
<th>Reading Overhead (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>756.048</td>
<td>772.624</td>
<td>16.576</td>
</tr>
<tr>
<td>2000</td>
<td>2.059.784</td>
<td>2.076.360</td>
<td>16.576</td>
</tr>
<tr>
<td>4000</td>
<td>3.365.104</td>
<td>3.381.680</td>
<td>16.576</td>
</tr>
<tr>
<td>6000</td>
<td>4.667.936</td>
<td>4.684.512</td>
<td>16.576</td>
</tr>
<tr>
<td>8000</td>
<td>5.975.040</td>
<td>5.991.616</td>
<td>16.576</td>
</tr>
<tr>
<td>10000</td>
<td>7.290.096</td>
<td>7.306.672</td>
<td>16.576</td>
</tr>
<tr>
<td>12000</td>
<td>8.581.280</td>
<td>8.597.856</td>
<td>16.576</td>
</tr>
<tr>
<td>14000</td>
<td>9.904.752</td>
<td>9.921.328</td>
<td>16.576</td>
</tr>
<tr>
<td>16000</td>
<td>11.196.104</td>
<td>11.212.680</td>
<td>16.576</td>
</tr>
<tr>
<td>18000</td>
<td>12.486.992</td>
<td>12.503.568</td>
<td>16.576</td>
</tr>
<tr>
<td>20000</td>
<td>13.826.544</td>
<td>13.843.296</td>
<td>16.752</td>
</tr>
<tr>
<td>21077</td>
<td>12.572.424</td>
<td>12.589.000</td>
<td>16.576</td>
</tr>
<tr>
<td>22000</td>
<td>15.116.512</td>
<td>15.133.088</td>
<td>16.576</td>
</tr>
<tr>
<td>22797</td>
<td>14.991.536</td>
<td>15.008.112</td>
<td>16.576</td>
</tr>
<tr>
<td>26000</td>
<td>17.763.656</td>
<td>17.780.232</td>
<td>16.576</td>
</tr>
<tr>
<td>28000</td>
<td>19.054.280</td>
<td>19.070.856</td>
<td>16.576</td>
</tr>
<tr>
<td>30000</td>
<td>20.345.728</td>
<td>20.362.304</td>
<td>16.576</td>
</tr>
<tr>
<td>34000</td>
<td>22.896.632</td>
<td>22.913.208</td>
<td>16.576</td>
</tr>
<tr>
<td>36000</td>
<td>24.184.720</td>
<td>24.201.296</td>
<td>16.576</td>
</tr>
<tr>
<td>38000</td>
<td>25.575.304</td>
<td>25.591.880</td>
<td>16.576</td>
</tr>
</tbody>
</table>

Table A.2: GXL Reading memory overhead
Appendix B   Processing Speed Test Results

B.1 Trimmed Mean Calculation

As explained in Chapter 4.4.4, unit tests were implemented to record the elapsed time while processing test graphs of different sizes. Since it turned out that the processing times varied between different executions of the same graph serialization or deserialization, the same test was automatically re-run 20 times and all the elapsed times were recorded. The results are presented in the following tables.

It can be examined, that the elapsed time varied between consecutive executions. The author assumes, that the reason for this behavior is the test machine environment. The tests were conducted on the authors computer (Intel Core2 CPU T7200 with 2GHz) which is not just running the Java Virtual Machine but also a variety of other processes that can be found in a standard end user operating system environment. The variations in the processor clock speed due to power saving techniques of the Intel Core2 CPU might have been another source for variations in processing speed.

A statistical method to obtain a robust estimator from a sample that includes extreme outliers (which are present in the test results) is the trimmed mean. To obtain valid figures describing the performance of the converter, outliers were truncated (trimmed) and an average was calculated. Since each test was run 20 times, each sample consists of 20 entries. From these 20 entries, the outlying 20% (4 entries) were truncated, resulting in the high and low 10% (2 entries) being omitted. The mean of the remaining 80% (16 entries) is the trimmed mean as it can be found in the following tables.
The following table contains measured elapsed times for GRAIL to GXL conversion with graphs containing up to 2,1077 graph elements (nodes + edges).

<table>
<thead>
<tr>
<th>Graph Elements</th>
<th>2</th>
<th>1000</th>
<th>2000</th>
<th>4000</th>
<th>6000</th>
<th>8000</th>
<th>10000</th>
<th>12000</th>
<th>14000</th>
<th>16000</th>
<th>18000</th>
<th>20000</th>
<th>21077</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>1</td>
<td>100</td>
<td>200</td>
<td>300</td>
<td>400</td>
<td>500</td>
<td>600</td>
<td>700</td>
<td>800</td>
<td>900</td>
<td>1000</td>
<td>1000</td>
<td>4856</td>
</tr>
<tr>
<td>Edges</td>
<td>1</td>
<td>100</td>
<td>200</td>
<td>300</td>
<td>400</td>
<td>500</td>
<td>600</td>
<td>700</td>
<td>800</td>
<td>900</td>
<td>1000</td>
<td>1000</td>
<td>16221</td>
</tr>
</tbody>
</table>

Measured times (milliseconds)

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>1000</th>
<th>2000</th>
<th>4000</th>
<th>6000</th>
<th>8000</th>
<th>10000</th>
<th>12000</th>
<th>14000</th>
<th>16000</th>
<th>18000</th>
<th>20000</th>
<th>21077</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trimmed Mean</td>
<td>11,13</td>
<td>159,19</td>
<td>272,31</td>
<td>407,44</td>
<td>583,06</td>
<td>654,19</td>
<td>843,19</td>
<td>953,81</td>
<td>1169</td>
<td>1260,5</td>
<td>1471,19</td>
<td>1199,06</td>
<td></td>
</tr>
</tbody>
</table>
The following table contains measured elapsed times for GRAIL to GXL conversion with graphs containing between 22000 and 40000 graph elements (nodes + edges).

<table>
<thead>
<tr>
<th>Graph Elements</th>
<th>22000</th>
<th>22797</th>
<th>24000</th>
<th>26000</th>
<th>28000</th>
<th>30000</th>
<th>32000</th>
<th>34000</th>
<th>36000</th>
<th>38000</th>
<th>40000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>11000</td>
<td>6186</td>
<td>12000</td>
<td>13000</td>
<td>14000</td>
<td>15000</td>
<td>16000</td>
<td>17000</td>
<td>18000</td>
<td>19000</td>
<td>20000</td>
</tr>
<tr>
<td>Edges</td>
<td>11000</td>
<td>16611</td>
<td>12000</td>
<td>13000</td>
<td>14000</td>
<td>15000</td>
<td>16000</td>
<td>17000</td>
<td>18000</td>
<td>19000</td>
<td>20000</td>
</tr>
<tr>
<td>Measured times</td>
<td>3247</td>
<td>2387</td>
<td>2803</td>
<td>3623</td>
<td>3722</td>
<td>4592</td>
<td>4245</td>
<td>4501</td>
<td>5051</td>
<td>5487</td>
<td>6018</td>
</tr>
<tr>
<td>(milliseconds)</td>
<td>1659</td>
<td>1604</td>
<td>2067</td>
<td>2058</td>
<td>2032</td>
<td>2376</td>
<td>3040</td>
<td>2847</td>
<td>2862</td>
<td>3228</td>
<td>2684</td>
</tr>
<tr>
<td></td>
<td>1703</td>
<td>1249</td>
<td>1797</td>
<td>1700</td>
<td>1795</td>
<td>1948</td>
<td>2476</td>
<td>2376</td>
<td>2571</td>
<td>2592</td>
<td>2506</td>
</tr>
<tr>
<td></td>
<td>1701</td>
<td>1149</td>
<td>1821</td>
<td>1746</td>
<td>2056</td>
<td>2262</td>
<td>2204</td>
<td>2073</td>
<td>2640</td>
<td>2845</td>
<td>2925</td>
</tr>
<tr>
<td></td>
<td>1632</td>
<td>1332</td>
<td>1554</td>
<td>1721</td>
<td>1793</td>
<td>2041</td>
<td>2440</td>
<td>2347</td>
<td>2610</td>
<td>2468</td>
<td>2632</td>
</tr>
<tr>
<td></td>
<td>1458</td>
<td>1254</td>
<td>1309</td>
<td>1651</td>
<td>1771</td>
<td>1760</td>
<td>2063</td>
<td>2331</td>
<td>2171</td>
<td>3042</td>
<td>2353</td>
</tr>
<tr>
<td></td>
<td>2076</td>
<td>1135</td>
<td>1326</td>
<td>1792</td>
<td>1700</td>
<td>1865</td>
<td>2297</td>
<td>2425</td>
<td>2284</td>
<td>297</td>
<td>3039</td>
</tr>
<tr>
<td></td>
<td>1617</td>
<td>1061</td>
<td>1545</td>
<td>1818</td>
<td>1715</td>
<td>1976</td>
<td>2369</td>
<td>2133</td>
<td>2302</td>
<td>2544</td>
<td>2279</td>
</tr>
<tr>
<td></td>
<td>1451</td>
<td>1304</td>
<td>1520</td>
<td>1430</td>
<td>1986</td>
<td>2088</td>
<td>1770</td>
<td>2039</td>
<td>2045</td>
<td>2719</td>
<td>2753</td>
</tr>
<tr>
<td></td>
<td>1451</td>
<td>1196</td>
<td>1736</td>
<td>1592</td>
<td>1682</td>
<td>2389</td>
<td>2067</td>
<td>2497</td>
<td>2156</td>
<td>2050</td>
<td>2439</td>
</tr>
<tr>
<td></td>
<td>1433</td>
<td>1429</td>
<td>1845</td>
<td>1864</td>
<td>1767</td>
<td>1661</td>
<td>2530</td>
<td>2259</td>
<td>2343</td>
<td>2926</td>
<td>2200</td>
</tr>
<tr>
<td></td>
<td>1395</td>
<td>1402</td>
<td>1617</td>
<td>1504</td>
<td>1660</td>
<td>1950</td>
<td>1972</td>
<td>2255</td>
<td>2403</td>
<td>2502</td>
<td>2554</td>
</tr>
<tr>
<td></td>
<td>1527</td>
<td>1365</td>
<td>1870</td>
<td>1736</td>
<td>1650</td>
<td>1876</td>
<td>2126</td>
<td>2228</td>
<td>2627</td>
<td>2427</td>
<td>2810</td>
</tr>
<tr>
<td></td>
<td>1275</td>
<td>1386</td>
<td>1512</td>
<td>1664</td>
<td>1563</td>
<td>2178</td>
<td>2527</td>
<td>2251</td>
<td>2073</td>
<td>2374</td>
<td>3923</td>
</tr>
<tr>
<td></td>
<td>1330</td>
<td>1329</td>
<td>1824</td>
<td>2109</td>
<td>1620</td>
<td>1649</td>
<td>2165</td>
<td>2015</td>
<td>2233</td>
<td>2451</td>
<td>2663</td>
</tr>
<tr>
<td></td>
<td>1790</td>
<td>1558</td>
<td>1704</td>
<td>1527</td>
<td>1630</td>
<td>2085</td>
<td>2170</td>
<td>1983</td>
<td>2525</td>
<td>3072</td>
<td>2591</td>
</tr>
<tr>
<td></td>
<td>1525</td>
<td>1037</td>
<td>1788</td>
<td>2120</td>
<td>1932</td>
<td>1759</td>
<td>1995</td>
<td>2469</td>
<td>2675</td>
<td>2942</td>
<td>2565</td>
</tr>
<tr>
<td></td>
<td>1530</td>
<td>1148</td>
<td>1607</td>
<td>1868</td>
<td>1785</td>
<td>2209</td>
<td>2441</td>
<td>2359</td>
<td>2454</td>
<td>2589</td>
<td>2673</td>
</tr>
<tr>
<td></td>
<td>1583</td>
<td>1430</td>
<td>1700</td>
<td>1607</td>
<td>2009</td>
<td>2207</td>
<td>1990</td>
<td>2277</td>
<td>2437</td>
<td>3091</td>
<td>2886</td>
</tr>
<tr>
<td></td>
<td>1621</td>
<td>1225</td>
<td>1767</td>
<td>1992</td>
<td>1792</td>
<td>1860</td>
<td>2393</td>
<td>2036</td>
<td>2200</td>
<td>2145</td>
<td>2775</td>
</tr>
</tbody>
</table>

Table B.2: GXL Writing measured times (Part 2)

The following table contains measured elapsed times for GRAIL to GXL conversion with graphs containing between 22000 and 40000 graph elements (nodes + edges).
The following table contains measured elapsed times for GXL to GRAIL conversion with graphs containing up to 21077 graph elements (nodes + edges).

<table>
<thead>
<tr>
<th>Graph Elements</th>
<th>2</th>
<th>2000</th>
<th>4000</th>
<th>6000</th>
<th>8000</th>
<th>10000</th>
<th>12000</th>
<th>14000</th>
<th>16000</th>
<th>18000</th>
<th>20000</th>
<th>21077</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>1</td>
<td>1000</td>
<td>2000</td>
<td>3000</td>
<td>4000</td>
<td>5000</td>
<td>6000</td>
<td>7000</td>
<td>8000</td>
<td>9000</td>
<td>10000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1000</td>
<td>2000</td>
<td>3000</td>
<td>4000</td>
<td>5000</td>
<td>6000</td>
<td>7000</td>
<td>8000</td>
<td>9000</td>
<td>10000</td>
<td>16221</td>
</tr>
<tr>
<td>Edges</td>
<td>214</td>
<td>152</td>
<td>1957</td>
<td>1483</td>
<td>1656</td>
<td>1952</td>
<td>2069</td>
<td>2040</td>
<td>2474</td>
<td>3270</td>
<td>2822</td>
<td>4022</td>
</tr>
<tr>
<td></td>
<td>66</td>
<td>328</td>
<td>542</td>
<td>751</td>
<td>949</td>
<td>850</td>
<td>1323</td>
<td>1445</td>
<td>1250</td>
<td>1491</td>
<td>1779</td>
<td>1571</td>
</tr>
<tr>
<td></td>
<td>65</td>
<td>359</td>
<td>367</td>
<td>601</td>
<td>660</td>
<td>898</td>
<td>819</td>
<td>1095</td>
<td>1262</td>
<td>1700</td>
<td>1453</td>
<td>1361</td>
</tr>
<tr>
<td></td>
<td>67</td>
<td>217</td>
<td>304</td>
<td>520</td>
<td>600</td>
<td>741</td>
<td>1093</td>
<td>1066</td>
<td>1254</td>
<td>1329</td>
<td>1773</td>
<td>1339</td>
</tr>
<tr>
<td></td>
<td>66</td>
<td>230</td>
<td>348</td>
<td>438</td>
<td>588</td>
<td>647</td>
<td>766</td>
<td>1155</td>
<td>1199</td>
<td>1451</td>
<td>1396</td>
<td>1320</td>
</tr>
<tr>
<td></td>
<td>65</td>
<td>200</td>
<td>375</td>
<td>545</td>
<td>635</td>
<td>819</td>
<td>778</td>
<td>963</td>
<td>990</td>
<td>1403</td>
<td>1326</td>
<td>1255</td>
</tr>
<tr>
<td></td>
<td>75</td>
<td>184</td>
<td>269</td>
<td>481</td>
<td>661</td>
<td>590</td>
<td>771</td>
<td>1054</td>
<td>982</td>
<td>1341</td>
<td>1625</td>
<td>1245</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>184</td>
<td>272</td>
<td>539</td>
<td>506</td>
<td>829</td>
<td>1027</td>
<td>941</td>
<td>1216</td>
<td>1213</td>
<td>1382</td>
<td>1364</td>
</tr>
<tr>
<td></td>
<td>70</td>
<td>249</td>
<td>281</td>
<td>392</td>
<td>550</td>
<td>652</td>
<td>911</td>
<td>1090</td>
<td>1164</td>
<td>1198</td>
<td>1388</td>
<td>1244</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>185</td>
<td>316</td>
<td>430</td>
<td>526</td>
<td>604</td>
<td>949</td>
<td>1078</td>
<td>1137</td>
<td>1307</td>
<td>1372</td>
<td>1276</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>170</td>
<td>278</td>
<td>535</td>
<td>625</td>
<td>602</td>
<td>730</td>
<td>1078</td>
<td>1194</td>
<td>1206</td>
<td>1396</td>
<td>1218</td>
</tr>
<tr>
<td></td>
<td>66</td>
<td>180</td>
<td>277</td>
<td>477</td>
<td>491</td>
<td>607</td>
<td>928</td>
<td>877</td>
<td>1241</td>
<td>1182</td>
<td>1576</td>
<td>1251</td>
</tr>
<tr>
<td></td>
<td>75</td>
<td>170</td>
<td>286</td>
<td>442</td>
<td>560</td>
<td>710</td>
<td>736</td>
<td>1239</td>
<td>1077</td>
<td>1287</td>
<td>1198</td>
<td>1188</td>
</tr>
<tr>
<td></td>
<td>63</td>
<td>168</td>
<td>259</td>
<td>474</td>
<td>547</td>
<td>650</td>
<td>880</td>
<td>896</td>
<td>1393</td>
<td>1257</td>
<td>1458</td>
<td>1386</td>
</tr>
<tr>
<td></td>
<td>63</td>
<td>160</td>
<td>284</td>
<td>450</td>
<td>506</td>
<td>603</td>
<td>710</td>
<td>907</td>
<td>1179</td>
<td>1097</td>
<td>1492</td>
<td>1180</td>
</tr>
<tr>
<td></td>
<td>122</td>
<td>169</td>
<td>279</td>
<td>489</td>
<td>478</td>
<td>783</td>
<td>855</td>
<td>1060</td>
<td>987</td>
<td>1110</td>
<td>1193</td>
<td>1363</td>
</tr>
<tr>
<td></td>
<td>62</td>
<td>169</td>
<td>270</td>
<td>416</td>
<td>517</td>
<td>612</td>
<td>967</td>
<td>1128</td>
<td>1198</td>
<td>1193</td>
<td>1476</td>
<td>1156</td>
</tr>
<tr>
<td></td>
<td>66</td>
<td>166</td>
<td>283</td>
<td>419</td>
<td>609</td>
<td>683</td>
<td>728</td>
<td>847</td>
<td>988</td>
<td>1245</td>
<td>1525</td>
<td>1310</td>
</tr>
<tr>
<td></td>
<td>68</td>
<td>160</td>
<td>284</td>
<td>500</td>
<td>538</td>
<td>723</td>
<td>768</td>
<td>1101</td>
<td>1194</td>
<td>1233</td>
<td>1144</td>
<td>1248</td>
</tr>
<tr>
<td></td>
<td>70</td>
<td>159</td>
<td>273</td>
<td>437</td>
<td>580</td>
<td>630</td>
<td>757</td>
<td>871</td>
<td>938</td>
<td>1348</td>
<td>1572</td>
<td>1282</td>
</tr>
</tbody>
</table>

| Trimmed Mean   | 67,13 | 195,56 | 298,56 | 486,06 | 575,5 | 696,44 | 858,44 | 1045,5 | 1158,13 | 1292,75 | 1463 | 1290,63 |

The following table contains measured elapsed times for GXL to GRAIL conversion with graphs containing up to 21077 graph elements (nodes + edges).
The following table contains measured elapsed times for GXL to GRAIL conversion with graphs containing between 22,000 and 40,000 graph elements (nodes + edges).

<table>
<thead>
<tr>
<th>Graph Elements</th>
<th>22000</th>
<th>22797</th>
<th>24000</th>
<th>26000</th>
<th>28000</th>
<th>30000</th>
<th>32000</th>
<th>34000</th>
<th>36000</th>
<th>38000</th>
<th>40000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>11000</td>
<td>6186</td>
<td>12000</td>
<td>13000</td>
<td>14000</td>
<td>15000</td>
<td>16000</td>
<td>17000</td>
<td>18000</td>
<td>19000</td>
<td>20000</td>
</tr>
<tr>
<td>Edges</td>
<td>11000</td>
<td>16611</td>
<td>12000</td>
<td>13000</td>
<td>14000</td>
<td>15000</td>
<td>16000</td>
<td>17000</td>
<td>18000</td>
<td>19000</td>
<td>20000</td>
</tr>
<tr>
<td>Measured times (milliseconds)</td>
<td>2929</td>
<td>3727</td>
<td>3626</td>
<td>4253</td>
<td>3742</td>
<td>4142</td>
<td>3891</td>
<td>4164</td>
<td>4197</td>
<td>4475</td>
<td>4486</td>
</tr>
<tr>
<td></td>
<td>1973</td>
<td>1722</td>
<td>2219</td>
<td>2830</td>
<td>2633</td>
<td>2827</td>
<td>3200</td>
<td>3318</td>
<td>3620</td>
<td>3489</td>
<td>3735</td>
</tr>
<tr>
<td></td>
<td>1878</td>
<td>1560</td>
<td>1955</td>
<td>2484</td>
<td>3007</td>
<td>2574</td>
<td>3014</td>
<td>2857</td>
<td>2759</td>
<td>3408</td>
<td>3475</td>
</tr>
<tr>
<td></td>
<td>1700</td>
<td>1594</td>
<td>1828</td>
<td>2235</td>
<td>2339</td>
<td>2499</td>
<td>2960</td>
<td>2774</td>
<td>2957</td>
<td>3208</td>
<td>3345</td>
</tr>
<tr>
<td></td>
<td>1637</td>
<td>1611</td>
<td>2026</td>
<td>1582</td>
<td>2429</td>
<td>2609</td>
<td>2716</td>
<td>3228</td>
<td>2996</td>
<td>3173</td>
<td>3544</td>
</tr>
<tr>
<td></td>
<td>2051</td>
<td>1293</td>
<td>2108</td>
<td>2297</td>
<td>1898</td>
<td>2589</td>
<td>2480</td>
<td>2564</td>
<td>2915</td>
<td>2326</td>
<td>2852</td>
</tr>
<tr>
<td></td>
<td>1940</td>
<td>1329</td>
<td>1544</td>
<td>1596</td>
<td>2169</td>
<td>2635</td>
<td>2032</td>
<td>2506</td>
<td>2969</td>
<td>3670</td>
<td>3398</td>
</tr>
<tr>
<td></td>
<td>1375</td>
<td>1414</td>
<td>2129</td>
<td>1905</td>
<td>2532</td>
<td>2387</td>
<td>2330</td>
<td>3000</td>
<td>2870</td>
<td>2675</td>
<td>3223</td>
</tr>
<tr>
<td></td>
<td>1847</td>
<td>1330</td>
<td>1503</td>
<td>2063</td>
<td>1844</td>
<td>2227</td>
<td>2255</td>
<td>2043</td>
<td>3149</td>
<td>3136</td>
<td>3138</td>
</tr>
<tr>
<td></td>
<td>1842</td>
<td>1376</td>
<td>1907</td>
<td>1898</td>
<td>2120</td>
<td>2307</td>
<td>2605</td>
<td>2663</td>
<td>2302</td>
<td>2643</td>
<td>3143</td>
</tr>
<tr>
<td></td>
<td>1405</td>
<td>1369</td>
<td>1752</td>
<td>1669</td>
<td>2029</td>
<td>2349</td>
<td>2522</td>
<td>2427</td>
<td>2927</td>
<td>2301</td>
<td>2991</td>
</tr>
<tr>
<td></td>
<td>1534</td>
<td>1385</td>
<td>1727</td>
<td>2076</td>
<td>2012</td>
<td>2496</td>
<td>2484</td>
<td>2616</td>
<td>2705</td>
<td>2846</td>
<td>3121</td>
</tr>
<tr>
<td></td>
<td>1481</td>
<td>1343</td>
<td>1730</td>
<td>1846</td>
<td>2104</td>
<td>2388</td>
<td>2109</td>
<td>2527</td>
<td>2684</td>
<td>2728</td>
<td>2505</td>
</tr>
<tr>
<td></td>
<td>1630</td>
<td>1297</td>
<td>1703</td>
<td>2152</td>
<td>2093</td>
<td>1931</td>
<td>2311</td>
<td>2487</td>
<td>2639</td>
<td>2889</td>
<td>2829</td>
</tr>
<tr>
<td></td>
<td>1620</td>
<td>1230</td>
<td>1830</td>
<td>1962</td>
<td>1995</td>
<td>2293</td>
<td>2555</td>
<td>2616</td>
<td>2797</td>
<td>2382</td>
<td>3070</td>
</tr>
<tr>
<td></td>
<td>1596</td>
<td>1342</td>
<td>1829</td>
<td>1926</td>
<td>2088</td>
<td>2290</td>
<td>2259</td>
<td>2484</td>
<td>2223</td>
<td>2613</td>
<td>2612</td>
</tr>
<tr>
<td></td>
<td>1617</td>
<td>1394</td>
<td>1793</td>
<td>1925</td>
<td>1746</td>
<td>1957</td>
<td>2106</td>
<td>2390</td>
<td>2624</td>
<td>2588</td>
<td>2902</td>
</tr>
<tr>
<td></td>
<td>1616</td>
<td>1271</td>
<td>1939</td>
<td>1968</td>
<td>2115</td>
<td>2408</td>
<td>2115</td>
<td>2393</td>
<td>2342</td>
<td>2296</td>
<td>2945</td>
</tr>
<tr>
<td></td>
<td>1565</td>
<td>1275</td>
<td>1788</td>
<td>1621</td>
<td>2112</td>
<td>1968</td>
<td>2265</td>
<td>2571</td>
<td>2608</td>
<td>2472</td>
<td>2381</td>
</tr>
<tr>
<td></td>
<td>1830</td>
<td>1299</td>
<td>1860</td>
<td>1929</td>
<td>1787</td>
<td>2310</td>
<td>2282</td>
<td>2583</td>
<td>2244</td>
<td>2734</td>
<td>2934</td>
</tr>
</tbody>
</table>

Table B.4: GXL Reading measured times (Part 2)

The following table contains measured elapsed times for GXL to GRAIL conversion with graphs containing between 22,000 and 40,000 graph elements (nodes + edges).
Appendix C  Developer Usage Instructions

C.1  Integration of GXL Converter Source Code

The GXL converter consists of the GXL class as an entry point for the VizzAnalyzer or for unit tests. Therefore, as described in more detail in Chapter 4.3.1, the GXL class implements the Converter interface. Together with the already existing converters, the GXL converter is part of the package grail.converters and should consequently be placed in the corresponding folder at “...\grail\converters\GXL.java” in the file system. The rest of the classes, that the converter consists of, are part of the package grail.converters.gxl. These files are:

- SAXGXLReader.java
- SAXGXLSchemaWriter
- SAXGXLWriter

They should be placed in the folder “...\grail\converters\gxl”. The converter should now be accessible by VizzAnalyzer in the same way as the already existing converters. As described in Chapter 5.2.2, an appropriate extension in the class GraphLoadSave is required to access all features of the GXL converter, due to an incomplete Converter interface.

To enable the unit tests, the corresponding class file TestGXL.java has to be added to the folder “...\test\grailtest”. Some generic functionality used by the test cases is externalized into the class GRAILTestUtils. This class is also part of the above mentioned grail.converters.gxl package. Therefore, the class file has to be placed in the folder “...\grail\converters\gxl”.

C.2  Running GXL Converter Unit Tests

Several unit test methods are implemented within the TestGXL class. Their main purpose is testing overall functionality, processing speed and memory overhead while converting different graphs. They are specifically implemented with the goal in mind, to produce valid results for Chapter 4.4. The purposes of the different test methods are documented in the relevant subchapters. Some of these tests, especially the ones that measure memory overhead might take considerable time to run. The test results are displayed on the console and each test is intended to be run separately with the results presented in the end. If all tests are run at the same time, a huge amount of output would be sent to the console and the execution time might be very long. Therefore, all the test methods have the access modifier private by default.

If a certain test result is needed, the corresponding test method has to be identified and the access modifier has to be changed to public. Within each test method are certain values, like for example the pathnames to specific GML graph files used for testing, that might have to be modified. The purpose of those values is in that case well documented in the source code.

C.3  Adjusting DTD, Schema and Metaschema File References

Both the GXL graph file and its corresponding GXL graph schema file follow the generic GXL DTD. The reference location for the DTD file is online at "http://www.gupro.de/GXL/gxl-1.0.dtd".
By default, the GXL converter uses this location in the produced GXL and GXL schema documents:

```
<!DOCTYPE gxl SYSTEM "http://www.gupro.de/GXL/gxl-1.0.dtd"> ...
```

**Code C.1: Default GXL DTD reference**

In addition, each GXL graph schema file contains references to the GXL metaschema file, for which the reference location is "http://www.gupro.de/GXL/gxl-1.0.gxl". These references occur throughout the whole schema document in every type attribute.

```
<node id="GRAILGraph">
  <type xlink:href="http://www.gupro.de/GXL/gxl-1.0.gxl#GraphClass"/>
  ...
</node>

<node id="GRAILNode">
  <type xlink:href="http://www.gupro.de/GXL/gxl-1.0.gxl#NodeClass"/>
  ...
</node>
```

**Code C.2: Default GXL metaschema reference**

As shown above, the converter also uses the reference location for the metaschema file in produced documents. This has the advantage, that the documents are portable in the way that the references to the files can be resolved on every machine as long as they can be accessed via the internet. However, it is possible to direct the GXL converter to write local directory references to the DTD and metaschema file into both the GXL graph and graph schema file. The following piece of code demonstrates this functionality.

```
... GXL gxl = new GXL();
gxl.setWriteLocalDTDandSchemaReferences(true);
gxl.toGXL(...)
```

**Code C.3: Enabling Local DTD and Schema References**

It can for example be used in the method `TestGXL.writeGraphToGXLFile`, which will cause all test graphs to be written with local file references. This allows using a local copy of the files “gxl-1.0.dtd” and “gxl-1.0.gxl” located in the same directory as the actual graph and graph schema file. At least the DTD file has to be reachable via the given reference during import of the graph with the GXL converter, since parsers usually access it even if they are not instructed to validate during parsing. Both DTD and metaschema file have to be reachable, if the GXL file validator by GUPRO (GUPRO 2005) is used to validate a graph and its schema.

More sophisticated ways to set these references might be of interest, which can be implemented by enhancing the above described setter method of the `GXL` class.