Master's thesis

Software Metrics in Static Program Analysis – A Formal Approach and its Application

Andreas Vogelsang
30 April 2010

Supervisors: Prof. Dr. Wolfgang Reif
             Prof. Dr. Alexander Knapp

External advisors: Dr. Ralf Huuck
                  Dr. Ansgar Fehnker
# Contents

1 Introduction 2

2 Related Work 5

3 Preliminaries 8
   3.1 Software Metrics 8
      3.1.1 Introduction 8
      3.1.2 Relevance 9
      3.1.3 Problems 12
   3.2 Static Program Analysis 13
      3.2.1 Introduction 13
      3.2.2 Goanna 14
   3.3 Software Metrics in Static Program Analysis 16

4 Definition of Metrics 18
   4.1 GMSL Language 18
      4.1.1 Syntax 19
      4.1.2 Semantics 21
   4.2 GMSL in Goanna 32
      4.2.1 Metric Definitions in Goanna 32
      4.2.2 Goanna’s Metric Library 33

5 Computation of Metrics 35
   5.1 GMSL Interpretation 35
   5.2 Goanna GMSL Interpreter 36
      5.2.1 Architecture 36
      5.2.2 Specifics 36
      5.2.3 Correctness 43
# Contents

6 **Visualization of Metrics**  
6.1 Output and Persistence of Metric Values ................. 44  
6.2 Visualization Views and Generic Charts .................. 45  

7 **Results**  
7.1 Performance ........................................ 52  
7.2 Validation of Results ................................... 54  

8 **Summary**  
8.1 Conclusion ............................................. 57  
8.2 Future Work ........................................... 58
Abstract

Software metrics make an important contribution to the controlling of software projects. To assure the quality of software, and especially its architecture, it is important to keep track of metric values already at development time. This work presents an integration of metric computation in static program analysis. The three main challenges are the definition of metrics, the computation of metrics, and the visualization of metrics. The work announces a fairly generic and formal approach to face these three challenges, but also shows its implementation in Goanna, a static program analyser, developed by the National ICT Australia. One key solution will be the introduction of GMSL, a language to define metrics on an abstract level.
I am heartily thankful to my advisors Dr. Ralf Huuck and Dr. Ansgar Fehnker from the National ICT Australia (NICTA) for giving me the opportunity to develop this work within the GOANNA project. Their advices and helpfulness contributed a lot to the success of this work. Thanks to the whole GOANNA team for supporting me and making me feel welcome.

I would also like to thank my supervisors Prof. Dr. Wolfgang Reif and Prof. Dr. Alexander Knapp from the University of Augsburg, whose guidance helped me greatly with the handling of semantics.

I am grateful to Michael Vistein from the University of Augsburg for doing all the paperwork while I was abroad and always offering me great advices on my actual work.

Lastly, I offer my regards and blessings to Sebastian Stey and Linda Hagemann who, reviewed my work and gave me very helpful suggestions on it.
Chapter 1

Introduction

There is no doubt that software development nowadays is a really complex and complicated matter. The software crisis in 1968 revealed that there is a strong need for mature processes and methodologies, which support software engineers to master the software projects of the future. The effect and the success of the so far established methods are open to dispute. However, current statistics are speaking for themselves. The 2009 Standish Group CHAOS Report [Grob] states that still 24% of software projects fail, meaning they are cancelled prior to completion or delivered and never used. While 32% of software projects are actually successful in the sense that they are delivered on time, on budget, with required features and functions, the majority of 44% are so called challenged software projects. These projects are late, over budget, and/or with less than the required features and functions. Even if the actual numbers of this report are controversially discussed in public, they reveal that the estimation of time and costs for software projects is hardly mastered.

Another characteristic of modern software projects is that almost no project is developed from scratch. Every new software has to provide or use certain interfaces to other systems. Legacy systems are involved frequently or have to be replaced. And even extensions of present software systems are developed as independent software projects. This means, systems that have to be developed by someone can be based on other systems, which have been developed by someone else.

The first paragraph suggests two main facts. Firstly, that the controlling
of software projects is a crucial activity in order to avoid them to deviate from the plan. Secondly, it shows that reliable figures about the quality of existing software can avoid misestimations, for example, in feasibility studies. The second paragraph reveals the importance of high quality architecture that supports maintainability, extensibility, and low complexity. Together they depict the need for methods to control the quality of software architectures as well as quality of software in general.

Tom DeMarco states in his book *Controlling Software Projects: Management, Measurement and Estimation*:

“You can’t control what you can’t measure.” [deM86]

According to this statement, a successful controlling relies on quantitative measurements. Such metrics make a contribution to the assurance of high software quality. However, codebases nowadays are really large and quality assurance is supposed to attend the development process continuously. That means, it demands great effort to control these large projects manually. Hence, a successful controlling should be tool supported to decrease the costs.

This work will introduce an approach to define, compute, and visualize software metrics in static program analysis. These software metrics are basic indicators of the quality of software architectures and enable a substantiated controlling of it. It will be shown how these metrics can be provided automatically and continuously during the development process. Therefore, a functional metric specification language will be introduced that allows the definition of user-defined metrics. A reference implementation of an interpreter of this language will also be described. It allows to automatically compute the values of the defined metrics for a given program. The visualization of the computed metric values will be based on a generic visualization model, which will be introduced in this work.

This work will not give any recommendation or proof which metrics are reasonable or particularly important. It will also not state how the computed metric values should be judged or when a metric value indicates good or poor quality. There will be some references in the next chapter to work that deals with such questions.

**Outline** After this introduction of the work’s topic, the second chapter will give an overview of related work that has already been done in affiliated fields. The third chapter introduces the preliminaries that are necessary to understand the approaches and solutions of the following sections. This
contains the basics of software metrics as well as the basics of static program analysis and especially GOANNA. The following three chapters 4, 5, and 6 cover the actual solutions in the areas of metric definition, metric computation, and metric visualization. Chapter 4, therefore, introduces the Goanna Metric Specification Language (GMSL) with its syntax, semantics, and its integration in GOANNA. Chapter 5 presents the interpretation and computation of GMSL expressions as well as some specifics of GOANNA’s implementation. Chapter 6 introduces a generic visualization model for metrics and describes an implementation of it. Chapter 7 then overviews the observations that have been made while testing the implementation on a real-world project. Chapter 8 summarizes all results and gives an outlook to following future work.
Chapter 2

Related Work

This work integrates software metrics into static program analysis. There is a variety of tools for both areas and even some of them already combine these fields. I will introduce a selection of tools that cover different aspects of this work’s subject and depict the similarities and differences to this work.

ODASA [Sem] is a commercial software assets analyser developed by a company called Semmle Limited. ODASA puts all software artefacts — not only the source code — into a software repository and provides a query engine for that repository. This query engine implements .QL, an object-oriented query language, used to retrieve data from relational databases. Semmle Limited develops .QL based on their company’s proprietary technology. The language allows the user to query the software repository for instances that fulfill the specified query properties. An example of a .QL query is the following:

```
1 from Project a, Project b
2 where a.dependsOn(b)
3 select a, b
```

This query finds the dependencies between projects. The types of the variables used in a query (e.g. Project) are user-defined object-oriented classes. The user can define new properties, like dependsOn, within these classes. These queries can be used to search the software for bottlenecks or other quality flaws. They can also be integrated into periodical reports that summarize the current results of the queries. In contrast to this work, ODASA supports a selective approach, where the user has to specify what he is looking for. ODASA can deal with Java, C/C++ and .NET languages.
Coverity Architecture Analysis [Cov] is part of a commercial static program analyser developed by Coverity. The main focus of the product is the bug detection for C/C++ and Java programs. The architecture analysis additionally aids the developer with the maintenance of architectural integrity. Therefore, it offers a range of visualizations, which mainly illustrate dependencies within a system as well as some predefined metrics with focus on complexity. Its Software DNA Map reveals accumulations of complexity or cyclic dependencies. Coverity Architecture Analysis is strongly focused on architectural analysis and offers very specific measures of a program. In comparison to the approach of this thesis, the user of Coverity Architecture Analysis is bound to the offered measures and cannot define his own metrics.

Klocwork Insight [Klo] is a commercial source code analysis product suite for Java, C/C++, and C# code developed by Klocwork. Its Integration Build Reporting and Metrics module enables the user to create reports based on metrics, applied to his source code. Klocwork Insight offers over 100 predefined metrics, which are derived directly from the program that is supposed to be analysed. It also offers to define custom metrics, but Klocwork Insight is not able to compute their values automatically on a program. Instead, the definition of a user-defined metric is given by a file that contains a listing with the actual metric values for all metric instances (e.g. for all classes). This file that contains the metric values must be provided manually. The metrics can be integrated in reports, which can be visualized afterwards. Klocwork Insight provides a set of diagram types to use for the visualization of reports.

NDepend [NDe] is a Visual Studio tool that helps the user to manage complex .NET codebases. With NDepend a series of standard and custom rules are continuously checked in Visual Studio with only low performance costs. NDepend considers the code as a database and the user can write Code Query Language (CQL) statements to query and check some assertions on this database. CQL as a language offers to query the code by accessing certain properties of it. Metric values in NDepend are represented by such properties. The user can, for example, query his code for all methods with a certain cyclomatic complexity or output a warning if there are methods that exceed a certain threshold. The resulting CQL query looks as follows:

```
1 WARN IF Count > 0 IN
2 SELECT METHODS
3 WHERE ILCyclomaticComplexity > 40
4 ORDER BY ILCyclomaticComplexity DESC
```

NDepend altogether offers 82 metrics, which can be integrated within queries.
There is no way of defining own additional metrics. NDepend provides a basic visualization to show the results of CQL queries. One of its highlights is a Treemap View that maps the values of the requested metrics to an area that is shown within the area of the superior scope instance. Other visualizations reveal dependencies or other specific metric values.

SonarJ is a software architecture management tool for systems written in Java and is based on static analysis. SonarJ is developed by the company hello2morrow and is free of charge for small projects (i.e. under 20,000 lines of code). Its main focus is to assure the consistency of the logical architecture of a system and its actual implementation. But besides that, SonarJ computes a bunch of metrics, including some reasonably complex metrics (e.g. Robert Martin’s metrics [Mar03]). SonarJ also allows the user to set thresholds for metric values. Developers who try to create code that exceeds a threshold receive a warning. SonarJ provides a basic histogram chart to visualize the development of metrics over time.

All of the mentioned tools can be partitioned into two different approaches. One approach is offering a query language that allows the user to query his code for instances with special properties. The code is represented in a repository database that enriches the code with computed values (including metric values). The second approach is the computation of metric values on the source code during the build process. These metric values can be visualized afterwards and must be evaluated and judged by the user. None of the introduced tools provide a mechanism that allows the user to define his own metrics that can be computed automatically by the tool. Therefore, the visualizations are partly very specific to the focus of the predefined metrics and measures.

The approach of this work will be more generic. It will enable the user to write his own metrics. The values of these metrics can be automatically computed by Goanna and the visualization is able to show the values of all defined metrics in a proper way.

There is also some work that focuses on certain metrics themselves and how they can improve software and its development, e.g. [NBZ06, MB03, FAW09, Els76, CSM79]. These references, however, are only for the interested reader, since this work will not focus on the reasonability of metrics or metric values. It will only introduce an approach to define, compute, and visualize them.
Chapter 3

Preliminaries

This chapter covers the preliminaries in the fields of software metrics and static program analysis that are necessary to understand the approaches and solutions of the following chapters. It also gives an introduction to GOANNA.

3.1 Software Metrics

The first section of this chapter will give the basics of software metrics. This especially contains an introduction with examples, an idea of the relevance of software metrics, but also a statement on problems and difficulties in the handling of them.

3.1.1 Introduction

A metric in general is a method to measure properties objectively. Metrics exist in many different engineering disciplines, where they measure properties of systems, products, processes, and many more. We are using common metrics everyday, for example, when expressing the property of heat as a number on the Celsius scale or the property of speed as the covered distance per time unit. Metrics have, amongst others, the advantage that their values are objective and can be compared to each other. In this function, metrics found their way into various theories, models, and methods, but also into laws, guidelines, and standards.
Software metrics are metrics that measure properties of software. The IEEE standard 1061 defines a software metric as

“A function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that affects its quality.

..."

The purpose of software metrics is to make assessments throughout the software life cycle as to whether the software quality requirements are being met. The use of software metrics reduces subjectivity in the assessment of software quality by providing a quantitative basis for making decisions about software quality. However, the use of software metrics does not eliminate the need for human judgement in software evaluations.” [IEE61]

This definition highlights the main aspects of software metrics: They make a statement about some quality attributes, they are quantitative, but they still have to be judged and evaluated by a human.

Within the field of software metrics you can further distinguish between different types of software metrics. Process metrics measure quality attributes of a process, e.g. lines of code per day, function points per person-day, or the CMMI measurements [CMM06]. Product metrics, on the contrary, measure quality attributes of the software product itself. This covers, amongst others, size metrics, complexity metrics, and quality metrics. Examples for product metrics are the number of lines of code, McCabe’s cyclomatic complexity [McC76], or the total number of detected bugs. Further types of software metrics are effort metrics, object-oriented metrics, or architecture metrics.

This work focuses on product metrics because they can be derived directly from the source code of a program, which is the object of investigation in static program analysis. Process metrics are only covered where they describe the development of a product metric over time.

### 3.1.2 Relevance

Software metrics play a major role in the controlling and steering of software projects. Metrics provide the basic drivers for any decision to change a software product or process. As we saw in the last section, there is a
great variety of software metrics and they also differ in the way of measuring them. The quantitative character of metrics offers an objective image of certain software properties. The metrics and their measures allow us to analyse them with mathematical and statistical methods and to connect them with other important figures. There is, for example, several work that has shown a correlation between certain complexity measures of a program and its likeliness to be faulty \cite{MB03, FAW09, NBZ06}. This underlines how relevant and helpful software metrics can be. According to the results of the named work, program parts with a high complexity should be avoided, or at least get more attention, since they are more likely to be faulty. There has also been some experimental work \cite{Els76, CSM79} which sought to correlate metrics with the ease of understanding a program or the ease of locating bugs. However, the results of that work are discussed controversially. The problem with such analyses is that it is hard to define if a program is easy to understand or good to reuse.

A scenario where software metrics are used as quality indicators is described in the standard ISO 9126 \cite{ISO01, ISO03a, ISO03b, ISO04}. Quality attributes can be structured into a Factor Criterion Metrics model (FCM-model). Such a model refines quality attributes into subcharacteristics. Software metrics are located at the bottom of this structure. Figure 3.1 shows the structure of an FCM-model. The arrows between the attributes define an is determined by relation. The FCM-model of ISO 9126 is shown in Figure 3.1 and has two levels of quality attributes. Quality attributes of the first level are Functionality, Reliability, Usability, Efficiency, Maintainability.
3.1. Software Metrics

and Portability. These quality attributes are further refined in the second level. The standard distinguishes between three kinds of metrics, which are

![Figure 3.2: FCM-model of ISO 9126](image)

used as quality indicators. They are shown in Figure 3.3 and will be described in the following. Internal quality relates to intermediate deliverables in the product construction, and thus can be already measured during the development. Internal quality and internal metrics allow predictions about the external quality, which describes the behaviour of a system. The quality in use describes whether the product meets the needs of a user in his specific context. This work will especially focus on internal quality and internal metrics, which are the basis for the external quality and the quality in use.

As we saw, metrics play an important role in the quality assurance of software products. And since quality assurance should be integrated into

![Figure 3.3: Kinds of metrics proposed in ISO 9126](image)
the entire software development process, it is important to have tools that support developers to measure the quality of their software already during the development.

### 3.1.3 Problems

While quality metrics have been accepted in other engineering disciplines long ago, software metrics are still discussed controversially. One reason for this is that metrics for the established sciences and engineering disciplines are based in the physical sciences, while software metrics are derived from human ingenuity. Another problem is that software quality in most cases is hard to define. Quality attributes like maintainability or portability are often a matter of opinion. That makes it difficult to find metrics that clearly measure the degree of maintainability or portability.

But not only the expressiveness of a special metric regarding a quality attribute is contended. There are also arguments about the existing software metrics themselves. Metrics like lines of code are highly dependent on programming languages and programming styles. Other metrics are defined on abstractions of software products, like the control flow graph, and thus cannot be applied directly to a program’s source code. While the first metric example was not abstract enough to make a statement about a software’s quality in general, the second metric example was too abstract to be derived directly from the source code.

A further example of this conflict between too abstract and too concrete metric definitions is McCabe’s cyclomatic complexity [McC76]. McCabe himself defined this metric on the control flow graph of a program. The cyclomatic complexity then is given as \( CC = e - n + 2p \), where \( e \) is the number of edges, \( n \) is the number of nodes and \( p \) is the number of strongly connected components in the control flow graph. As you can see, this definition cannot be applied directly to a program. You first have to extract the control flow graph. Another definition of the same metric is given by NDepend, a manufacturer of a metric computation tool. They define cyclomatic complexity as [NDe]:

\[
CC = 1 + \{ \text{number of following expressions found in a method} \} : \\
\quad \text{if} \mid \text{while} \mid \text{for} \mid \text{foreach} \mid \text{case} \mid \text{default} \mid \\
\quad \text{continue} \mid \text{goto} \mid \&\& \mid || \mid \text{catch} \mid \text{ternary operator} : \mid ??
\]

So their definition is dependent on concrete code constructs, which differ between programming languages. The consequence is that this definition
is only valid for the programming language C#. This work introduces an approach to define metrics on an abstract level, which allows the user, for example, to derive their definition directly from its mathematical abstraction. However, the metric definitions can be automatically applied directly to a program, independent from the used programming language.

Reliable software metrics and a support to collect their values will be a necessity for the evolution of software engineering from an art to an engineering discipline. This work especially focuses on the last mentioned problem and introduces a way of defining metrics in a generic way, without losing the ability to compute their values automatically for a given software product. This approach will not be outdated when there is clearness about which metrics finally are reasonable.

## 3.2 Static Program Analysis

This section introduces the basics of static program analysis in general and especially the static program analysis tool Goanna.

### 3.2.1 Introduction

Static Program Analysis is mainly used as a static test method for software systems to find bugs or anomalies, but it can also be used, for example, for compiler optimizations. *Static* in this case means that the software system is not being executed during a test. As a test method, static program analysis can only reveal the presence of bugs, but in general never prove the absence of them\(^1\). The subject on which a static program analysis is performed is in most cases either the source code or the object code of a program. Since a program does not need to be executed during a static program analysis, software can already be tested in an early stage of the development process, even when a program is not yet executable. This allows to find bugs and bottlenecks in programs very early and therefore reduces the development costs. Another advantage of static program analysis is that it can be largely performed automatically by tools. A human analysis of a program’s source code is often referred to as *program understanding* or *code review*.

Besides bug detection, static program analysis can also be used to check the conformity with some programming guidelines, find unsafe type casts, or

\(^1\)Actually static program analysis can show the absence of bugs for certain classes, e.g. null-pointer dereferencing or type safety
3.2. Static Program Analysis

... even to compute some metrics. For this purpose, the tools use different techniques like model checking, abstract interpretation, or data-flow analysis to implement the analysis. Most of these techniques are based on an abstraction of the actual program as a mathematical model. Bugs or other problems that have to be checked are encoded as properties of this mathematical model. The tools determine them to be true or false for the present abstraction of a program.

One abstraction of a program that plays an important role in static program analysis and especially in this work is the representation of a program’s source code as Abstract Syntax Tree (AST). The AST is a tree representation of a program’s syntax given by its source code. It is called abstract because not all syntactic constructs are exactly transformed to a node of the AST. Grouping parentheses, for example, are omitted because they are implicit in the tree structure. Figure 3.4 gives an example of an AST extracted from a program.

Figure 3.4: A program’s representation as source code and Abstract Syntax Tree (AST)

3.2.2 Goanna

Goanna [FHJ+07] is a tool for static program analysis of C/C++ programs and is based on model checking. Its main goal is to find bugs in programs at development time. The idea behind GOANNA is to extract the Control Flow Graph (CFG) of a program, transform it into a Kripke structure and use a model checker to check whether certain properties hold. An overview
of GOANNA’s main workflow is given by Figure 3.5. The principles of model checking will not be discussed in detail here and can be looked up in [CGP99].

Figure 3.5: GOANNA’s model checking approach for statically analysing C/C++ code [FHJ+07].

For the definition and model checking of properties, we need to specify what the propositions of the model are. As mentioned before, GOANNA performs model checking on the CFG of a program. So the nodes of the model are blocks of the program code. A block can be a single statement or a group of statements that are always executed together (i.e. there is no possibility that the control jumps into or out of this block).

GOANNA extracts the CFG out of the AST of a program. The propositions of the CFG nodes are special properties that a node fulfills. To differentiate between these properties of the CFG nodes and the properties of a whole program, which are checked by the model checker, the properties of the CFG nodes are called annotations.

The annotations of a node are determined by the membership of the node in a nodeset, which is returned by a function that queries the AST. For example, let \( xread \) be a nodeset that is returned by a query function \( read(x) \). This function returns all nodes of the AST in which a variable \( x \) is read. All nodes of the extracted CFG that contain a statement that is a member of \( xread \), will get \( read(x) \) as an annotation.
3.3. Software Metrics in Static Program Analysis

These annotations can be used to define properties on the whole program. If we extend the prior example by another annotation, which tags all CFG nodes that contain a declaration of variable $x$, we can define a temporal formula that postulates the existence of a node with annotation $\text{decl}(x)$ preceding every node with annotation $\text{read}(x)$. If the model fulfills this property, there are no read accesses to an undeclared variable $x$.

**Goanna** provides two languages to define properties and annotations. The definition of an annotation is written in the Goanna XPath Specification Language (GXSL) [VOR+09]. GXSL is very close to the well known XPath language [CD99] and enables the user to define queries on the AST.

The definition of properties is done in a language called Goanna Property Specification Language (GPSL) [VOR+09]. This language is basically an enrichment of Computation Tree Logic (CTL) [CES86] with additional property patterns, which make it more comfortable to define properties related to static program analysis.

While these generic languages allow any definition of properties and annotations, they are used in **Goanna** to define properties that represent the occurrence of a bug in a program.

**Goanna** as a tool is used like a compiler. It can be applied to a single source file and performs its analysis after compiling the file. Therefore, **Goanna** can easily be integrated into existing software projects by just replacing the used compiler with **Goanna**. Any detected error is output on the command line. In addition to the output of errors, **Goanna** also provides a counterexample trace, where the user can follow the control flow in which an error might occur. **Goanna** exists as a plugin for the Eclipse IDE as well as for Microsoft Visual Studio. These plugins are basically graphical interfaces for the command line version and enable the user to select files to analyse more easily.

### 3.3 Software Metrics in Static Program Analysis

As we saw, static program analysis examines the source code of a program to check it for certain properties in an early stage of development. So, when we speak about software metrics in static program analysis, we are speaking of metrics that are exclusively dependent on the source code. These metrics are quantitative figures, which characterize the program in several ways.

This definition limits the amount of metrics that can be handled in static
3.3. Software Metrics in Static Program Analysis

Program analysis. Metrics that can only be measured at runtime of a program, e.g. metrics of usability, metrics of performance, or metrics of requirements conformance, cannot be supported in static program analysis.

The essence of computing software metrics in static program analysis is basically counting occurrences of certain code constructs and afterwards performing some calculations on these figures. Thus, the design of the metric definition language described in this work, will depend on a code exploration mechanism. This mechanism has to provide functions that query the AST of a program and return all nodes that fulfill a certain property. GOANNA already implements an established mechanism to explore the source code, respectively the AST, of a program.

As we saw in Chapter 3.1.3 a good definition of a metric is a thin line between being too abstract and being too specific. To face this challenge, this work introduces a definition language in the next chapter. The purpose of this language is to enable the user to define metrics on an abstract level, but also enable an interpreter of this language to compute the defined metrics on a given program. GOANNA’s metric module will provide such an interpreter to compute metrics on C/C++ programs.
Chapter 4

Definition of Metrics

This chapter introduces GMSL, a functional metric definition language. The syntax and the semantics of the language will be given before its integration in GOANNA is presented.

4.1 GMSL Language

The Goanna Metric Specification Language (GMSL) is a definition language for software metrics in static program analysis. It provides a way to define metrics on an abstract level, but concrete enough to compute them on real programs.

A prerequisite for the use of GMSL is a query engine that returns nodes of the AST for which certain properties hold.

As mentioned in Chapter 3.2.2, GOANNA implements such an engine. Queries can be expressed in GOANNA’s GXSL language and allow the user to select certain nodes of the AST of a program. The queries are always evaluated on the entire AST but it is possible to pass parameters to the queries to refer to special nodes or properties of nodes in the AST. The result of a GXSL query that is applied to a program, is a set of AST nodes.

Most metrics are defined for a given scope. For example, a metric might be defined for the scope all_classes, which means that one metric value will be computed for each class. Other metrics are defined for scopes like
functions or namespaces. In GMSL you declare the scope of a metric in its definition. The metric values will be computed for every instance of the scope. The language handles two different types of variables, distinguished by their syntax. One type of variable represents in each case one node of the AST. The values of these variables are typically returned by a GXSL function and will be passed to further GXSL queries. Their syntax can be an arbitrary identifier (see GMSL grammar in Section 4.1.1). The other type of variable represents in each case a real number, either determined by the number of occurrences of certain nodes in the AST or by the aggregation of further query results. They can also be used to bind results of other defined metrics. These variables have an ‘@’ as prefix.

The actual definition of the metric then is a mathematical expression containing counting variables, queries and constant numbers.

### 4.1.1 Syntax

This section introduces the syntax of the GMSL language. It will be presented by its grammar, given in Extended Backus–Naur Form (EBNF). After the grammar, some examples of GMSL expressions are stated.

#### GMSL Grammar in EBNF

```plaintext
gmsl  = "METRIC" name scope [venv] definition ;
scope  = '( node "IN" function ')' ;
name  = ident ;
venv  = "WITH" vdecl (',' vdecl)* ;
vdecl  = var '=' binding ;
definition  = "DEF" expression ;
binding  = function | aggregator function "OVER" setindex ;
aggregator  = "SUM" | "MAX" | "MIN" | "PROD" ;
setindex  = node "IN" function ;
function  = ident [ '(' [ ident (',' ident)* ] ')' ] ;
expression  = var | function | num | expression op expression ;
op  = '+' | '-' | '*' | '/' ;
var  = '@' ident ;
node  = ident ;
num  = nat | real ;
nat  = ( '0' | ... | '9' )+ ;
real  = nat '.' nat ;
ident  = ( 'a' | 'b' | ... | 'Z' | '_' )+ ;
```

#### Examples

This section shows a few examples to familiarize with the syntax of GMSL. There are a few functions used in these examples, which are provided by GOANNA’s AST query library. If no library exists or some functions are
missing, you can provide user-defined AST queries, e.g. as GXSL expressions, to be used in metric definitions.

**Cyclomatic Complexity**

Cyclomatic Complexity of a function as defined in [NDe] is the number of branches in the control flow of a function increased by one. If we only consider one function, i.e. one strongly connected component, this definition is equal to McCabe’s definition [McC76], which defines the cyclomatic complexity as the number of linearly independent paths in the control flow of a function:

```
1 METRIC cc_per_f (f IN all_funs)
2 WITH @cn = all_cond_nodes(f)
3 DEF 1 + @cn
```

The metric will be computed for all nodes \( f \) returned by the AST query \( \text{all\_funs} \), which returns the corresponding AST node for every function of a given program. The metric value of \( f \) is determined by the number of conditional nodes in \( f \), given by \( \text{all\_cond\_nodes} \), increased by one.

**Efferent Coupling**

Efferent Coupling of a class as defined in [iSAb] is the number of classes that are called from a certain class:

```
1 METRIC efferent_coupling (c IN all_classes)
2 WITH @ce = dependencies(c)
3 DEF @ce
```

The metric will be computed for all nodes \( c \) returned by the AST query \( \text{all\_classes} \), which returns the corresponding AST node for every class of a given program. The metric value of \( c \) is determined by the number of dependencies \( c \) has. The AST query \( \text{dependencies}(c) \) returns a node for all classes that are called in class \( c \).

**Afferent Coupling**

Afferent Coupling of a class as defined in [iSAa] is the number of classes that call a certain class:

```
1 METRIC afferent_coupling (c IN all_classes)
2 WITH @ca = SUM dependency(g,c) OVER g IN all_classes
3 DEF @ca
```

The metric will be computed for all nodes \( c \) returned by the AST query \( \text{all\_classes} \). The metric value of \( c \) is determined by the sum of \( \text{dependency}(g,c) \),
applied to all nodes \( g \), returned by the AST query \(_{\text{all_classes}}\). The AST query \(_{\text{dependency}}(g, c)\) returns one node for class \( g \), if there is a function call in class \( g \) to class \( c \). That means, the sum is increased by one, if a node is being returned.

**Cohesion**

Cohesion of a class as defined in [BB04] is a measure of how strongly-related and focused the various responsibilities of a class are, depending on how many methods of a class access common fields or call common other methods:

\[
\text{Cohesion} = \frac{\text{SUM} \text{ directly-related} \text{ of methods} \text{ of class } c}{\text{METHODS} \text{ of class } c \times (\text{METHODS} \text{ of class } c - 1)}
\]

The metric will be computed for all nodes \( c \) returned by the AST query \(_{\text{all_classes}}\). The AST query \(_{\text{directly-related}}(m)\) returns a node for all methods of the same class that are directly related to method \( m \) (i.e. they both access a certain common field or they are both calling another common method of the class). If every method is directly related to all other methods, then the metric value is equal to 1. This last example also shows the implicit conversion of an AST query result bound directly to the counting variable \( @N \). The result of \(_{\text{methods_of_class}}(c)\) is implicitly converted from a set of nodes to a number that represents the cardinality of the set. We will have a closer look at this, when defining the semantics in the next section.

### 4.1.2 Semantics

The semantics of the GMSL language will be given as a denotational semantics. There are two reasons why it is a reasonable choice to define the semantics in a denotational way.

The first and more important reason is the mathematical character of a metric itself. The definition and computation of a metric can easily be seen as a mathematical function. When applied to a program or a fragment of a program, this function returns the value of the metric for that program (e.g. a real number). Thus, the semantics of a metric definition on an abstract level can be seen as follows:

\[
S[\cdot] : M \rightarrow \text{Prog} \rightarrow \mathbb{R}
\]

The second reason is the implementation of a GMSL interpreter in GOANNA. A denotational semantics can be implemented very intuitively because GOANNA
is written in a functional programming language. This second argument is, of course, much less important, since an implementation should, in no way, affect the definition of a semantics.

To define the actual semantics of the GMSL language, we have to introduce some fundamentals. The real numbers are denoted by \( \mathbb{R} \).

**Definition 4.1.1** The value of a symbolic syntax variable \( n \) is denoted by \( N[n] \in \mathbb{R} \).

**Definition 4.1.2** Let \( \text{Prog} \) be the set of all programs, given by their AST. We further denote the set of nodes, contained in the AST of \( p \in \text{Prog} \) by the dependent type \( \Pi p : \text{Prog}.\text{nodes}(p) \).

**Definition 4.1.3** Let \( \text{MDecl}^* \) be a set of metric declarations, given as GMSL expressions. We further denote the metric names of a set of metric definitions \( M \in \text{MDecl}^* \) by the dependent type \( \Pi M : \text{MDecl}^*.\text{names}(M) \). The metric name of a single metric \( m \in \text{MDecl} \) is given by the function \( (m) \).

**Definition 4.1.4** Let \( \text{Lib} \) be the set of library functions, given as functions \( \varsigma \in \text{Lib} : \text{FName} \to \Pi p : \text{Prog}.\text{nodes}(p)^\ast \to \mathbb{R}(\text{nodes}(p)) \). Library functions take a program and a list of nodes of this program as arguments and return a set of program nodes that fulfill a given property.

**Definition 4.1.5** Let \( \text{MEnv} \) be the set of metric environments, given as functions \( \mu \in \text{MEnv} : \text{MName} \to \Pi p : \text{Prog}.\text{nodes}(p) \to \mathbb{R} \). A metric environment maps a name of a metric to a function that maps some nodes of a given program to their corresponding metric values.

**Definition 4.1.6** Let \( \text{NEnv} \) be the set of node variable environments, given as functions \( \eta \in \text{NEnv} : \text{NIdent} \to \Pi p : \text{Prog}.\text{nodes}(p) \). A node environment maps syntactical node identifier to their corresponding AST node of a given program.

**Definition 4.1.7** Let \( \text{VEnv} \) be the set of counting variable environments, given as function \( \nu \in \text{VEnv} : \text{VIdent} \to \mathbb{R} \). Counting variable environments map syntactical counting variable identifier (e.g. @x) to their corresponding value.

**Definition 4.1.8** Let \( \text{Env} = \text{Lib} \times \text{MEnv} \times \text{NEnv} \times \text{VEnv} \) be the summary of all defined environments.
With this preliminaries we can now define the denotations for a GMSL program, which contains a set of metric definitions:

\[
P[-] : \Pi M : MDecl^* \Pi p : Prog. Lib \rightarrow [nodes(p) \times names(M) \rightarrow \mathbb{R}]
\]

\[
P[M]_{\varsigma} = C[M](M[M]_{\varsigma})
\]

\(P\) is the semantic function of a sequence of metric definitions \(M\) written in GMSL. Given a program \(p\) and a set of library functions, the semantic function returns a partial function that maps a scope node and a given metric to a real number. The actual evaluation of such expressions is divided into two parts. The first part contains the declarations of the metrics. The semantic function \(M\) converts the metric declarations into a metric environment, used in the second part, which performs the actual evaluation of the metric values. This second part is given by the semantic function \(C\).

\[
C[-] : \Pi M : MDecl^* . \Pi p : Prog. MEnv \rightarrow [nodes(p) \times names(M) \rightarrow \mathbb{R}]
\]

\[
C[M]_{\mu} = \bigcup_{m \in names(M)} \{ (n, m) \mapsto \mu(m)(p)(n) \}
\]

This semantic function \(C\) finally evaluates the metric expressions and builds the function that is returned by \(P\). A metric evaluation is just a lookup in the given metric environment, since all metric definitions are on hand in the environment as a kind of compiled metrics.

\[
M[-] : MDecl^* \rightarrow Lib \rightarrow MEnv
\]

\[
M[m_1 \ldots m_2]_{\varsigma} = M'[m_1](\varsigma, (M'[m_2](\varsigma, \ldots (M'[m_n](\varsigma, \emptyset)))\ldots))
\]

\[
M'[[-]] : MDecl \times MEnv \rightarrow MEnv
\]

\[
M'[m]_{(\varsigma, \mu)} = \mu [name(m) \mapsto S[m](\varsigma, \mu, \emptyset, \emptyset)]
\]

The semantic functions \(M\) and \(M'\) compile the given metric definitions into a metric environment. As we saw in Definition 4.1.5, a metric environment maps a metric, given by its name, to a function that maps nodes of a given program to real numbers. Thus, all information that are necessary for applying a metric definition to a program are contained in the metric environment. \(M\) creates and updates a metric environment step by step by mapping the
names of the declared metrics to a function that is returned by another semantic function $S$.

$$S[-] : MDecl \rightarrow (Env \rightarrow (\Pi p : \text{Prog}.\text{nodes}(p) \rightarrow \mathbb{R}))$$

$$S[\text{METRIC name (scope IN f) venv definition}] = \lambda p \in \text{Proj} . \lambda n \in G[f]((\varsigma, \eta))$$

The idea behind this is that a GMSL definition, given an initial environment, maps the environment to a function that takes a program and maps some nodes of this program to real numbers. These real numbers represent the values of the metric for the given nodes, which correspond to the instances of the metric’s scope.

All used semantic functions of subparts of the above definition are described below:

$$\text{upd}_V : \text{venv} \rightarrow Env \rightarrow (\text{Proj} \rightarrow Env)$$

$$\text{upd}_V(\text{WITH vdecl}_1, \ldots, \text{vdecl}_n)((\varsigma, \mu, \eta, \nu) =$$

$$\lambda p \in \text{Proj} . \text{upd}'_V(\text{vdecl}_1)(\varsigma, \mu, \eta, \ldots, \text{upd}'_V(\text{vdecl}_n)(\varsigma, \mu, \eta, \nu)(p)))$$

$$\text{upd}'_V : \text{vdecl} \rightarrow Env \rightarrow (\text{Proj} \rightarrow Env)$$

$$\text{upd}'_V(\text{@v = binding})(\varsigma, \mu, \eta, \nu) =$$

$$\lambda p \in \text{Proj} . ((\varsigma, \mu, \eta, \nu[@v \rightarrow B[binding]((\varsigma, \mu, \eta)(p))])$$

These two update functions manipulate $\nu \in \text{VEnv}$ in the way that declared counting variables are mapped to the semantics of their bindings.
4.1. GMSL Language

\[ B[-] : binding \rightarrow (Lib \times MEnv \times NEnv \rightarrow (Prog \rightarrow \mathbb{R})) \]

\[ B[f]((\zeta, \mu, \eta) = \lambda p \in Prog . F[f]((\zeta, \mu, \eta)(p)) \]

\[ B[SUM f OVER node IN g]((\zeta, \mu, \eta) = \lambda p \in Prog . \sum_{n \in G_g[\zeta, \eta](p)} F[f]((\zeta, \mu, \eta[\text{node} \mapsto n])(p)) \]

\[ B[PROD f OVER node IN g]((\zeta, \mu, \eta) = \lambda p \in Prog . \prod_{n \in G_g[\zeta, \eta](p)} F[f]((\zeta, \mu, \eta[\text{node} \mapsto n])(p)) \]

\[ B[MAX f OVER node IN g]((\zeta, \mu, \eta) = \lambda p \in Prog . \max_{n \in G_g[\zeta, \eta](p)} F[f]((\zeta, \mu, \eta[\text{node} \mapsto n])(p)) \]

\[ B[MIN f OVER node IN g]((\zeta, \mu, \eta)(p) = \lambda p \in Prog . \min_{n \in G_g[\zeta, \eta](p)} F[f]((\zeta, \mu, \eta[\text{node} \mapsto n])(p)) \]

A binding of a counting variable can either be a simple function or an aggregation over a set of numbers determined by the application of a function on the results of a nodeset, returned by another function. Simple functions in this case can be AST query functions from the library or the name of another metric. Thus, the semantics of a simple function as binding of a counting variable is determined by the semantic function \( F \), which basically distinguishes between these possibilities and returns a real number. For the aggregation, the function \( f \) will be applied to all results of the function \( g \) and afterwards the values of the returned real numbers will be aggregated. In the special case of the function \( g \), only AST query functions from the library are allowed because \( g \) must return a set of nodes that can be used afterwards in \( f \). Therefore, \( g \) is evaluated by the semantic function \( G \), whereas \( f \) is evaluated by \( F \) as supplied before.

\[ D[-] : definition \rightarrow (Env \rightarrow (Prog \rightarrow \mathbb{R})) \]

\[ D[DEF expression]((\zeta, \mu, \eta, \nu) = \lambda p \in Prog . E[expression]((\zeta, \mu, \eta, \nu)(p)) \]
\[ \mathcal{E}[\cdot] : \text{expression} \to (\text{Env} \to (\text{Prog} \to \mathbb{R})) \]

\[ \mathcal{E}[\text{env}](\varsigma, \mu, \eta, \nu) = \lambda p \in \text{Prog}. \mu(p) \]

\[ \mathcal{E}[\text{fun}(n_1, \ldots, n_k)](\varsigma, \mu, \eta, \nu) = \lambda p \in \text{Prog}. \mathcal{F}[\text{fun}(n_1, \ldots, n_k)](\varsigma, \mu, \eta)(p) \]

\[ \mathcal{E}[\text{fun}(n_1, \ldots, n_k)](\varsigma, \mu, \eta, \nu) = \lambda p \in \text{Prog} \cdot N(n) \]

\[ \mathcal{E}[\text{exp}_1 + \text{exp}_2](\varsigma, \mu, \eta, \nu) = \lambda p \in \text{Prog}. \mathcal{E}[\text{exp}_1](\varsigma, \mu, \eta, \nu)(p) + \mathcal{E}[\text{exp}_2](\varsigma, \mu, \eta, \nu)(p) \]

\[ \mathcal{E}[\text{exp}_1 - \text{exp}_2](\varsigma, \mu, \eta, \nu) = \lambda p \in \text{Prog}. \mathcal{E}[\text{exp}_1](\varsigma, \mu, \eta, \nu)(p) - \mathcal{E}[\text{exp}_2](\varsigma, \mu, \eta, \nu)(p) \]

\[ \mathcal{E}[\text{exp}_1 \cdot \text{exp}_2](\varsigma, \mu, \eta, \nu) = \lambda p \in \text{Prog}. \mathcal{E}[\text{exp}_1](\varsigma, \mu, \eta, \nu)(p) \cdot \mathcal{E}[\text{exp}_2](\varsigma, \mu, \eta, \nu)(p) \]

\[ \mathcal{E}[\text{exp}_1 / \text{exp}_2](\varsigma, \mu, \eta, \nu) = \lambda p \in \text{Prog}. \mathcal{E}[\text{exp}_1](\varsigma, \mu, \eta, \nu)(p)/\mathcal{E}[\text{exp}_2](\varsigma, \mu, \eta, \nu)(p) \]

An expression can either be a counting variable, a function, a constant number or a composition of expressions. If the expression is a counting variable, the semantics of it is just the semantics of the binding to which it is mapped in the counting variable environment. If the expression is a function, it can either be an AST query function from the library or another metric. The actual value of the expression thus is determined by \( \mathcal{F} \). Usage of a number as an expression is just mapped to the semantics of the number itself.

\[ \mathcal{F}[\cdot] : \text{function} \to (\text{Lib} \times \text{MEnv} \times \text{NEnv} \to (\text{Prog} \to \mathbb{R})) \]

\[ \mathcal{F}[\text{libfun}(n_1, \ldots, n_k)](\varsigma, \mu, \eta) = \lambda p \in \text{Prog}. |\mathcal{G}[\text{libfun}(n_1, \ldots, n_k)](\varsigma, \eta)(p)| \]

\[ \mathcal{F}[\text{metric}(n)](\varsigma, \mu, \eta) = \lambda p \in \text{Prog}. \mu(\text{metric})(p)(\eta(n)(p)) \]

This semantic function determines whether a function is an AST query function from the library or another metric. In both cases \( \mathcal{F} \) returns a real number. In the first case the result is the cardinality of the nodeset returned by the library function. In case of another metric as a function, the result is the value of that metric for the given node. The use of other metrics only allows exactly one parameter.
4.1. GMSL Language

\[ G[\text{libfun}(n_1, \ldots, n_k)](\varsigma, \eta) = \lambda p \in \text{Prog} \cdot \varsigma(\text{libfun}(p))(\eta(n_1)(p), \ldots, \eta(n_k)(p)) \]

\( G \) as semantic function determines the application of an AST query function from the library. The application is a lookup in the library for the called function, which is supplied with the actual parameters, that are stored in the node variable environment \( \eta \in \text{NEnv} \). The result of \( G \) is a set of nodes of the given AST.

Since GMSL, by its design, is based on queries on the AST of a program, we assume that the \( \text{Lib} \) is given prior. All necessary functions exist and are unique. The functions are entirely determined by their arguments. They apply a query to a program’s AST and return a set of nodes that fulfill the query.

**Example**

To check the validity of the defined semantics, we should have a look at an example. Let \( M \) be the following program of metric definitions:

```plaintext
1 METRIC cc_per_f (f IN all_funs)
2 WITH @cn = all_cond_nodes(f)
3 DEF 1 + @cn
4
5 METRIC avg_method_cc (c IN all_classes)
6 WITH @s = SUM cc_per_f(m) OVER m IN methods_of_class(c),
7     @n = methods_of_class(c)
8 DEF @s / @n
```

The program \( M \) contains two metric definitions. The first metric \( cc\_per\_f \) computes the cyclomatic complexity of a function as described in Section 4.1.1. The second metric uses the results of the first metric to compute the average cyclomatic complexity of the methods of a class. All function calls, except \( cc\_per\_f(m) \), are calls to AST query functions from the library.

We will apply these metric definitions to the following C++ program:

```plaintext
class Number {
    private:
        int n;
    public:
        Number(int number) { n = number; }
        void inc();
        void dec();
};
```

Software Metrics in Static Program Analysis — A Formal Approach and its Application
4.1. GMSL Language

```cpp
void Number::inc()
{
    n++;
}

void Number::dec()
{
    if (n>0) n--;
}

int main()
{
    return 0;
}
```

This C++ program consists of one class with two public methods and one constructor and a `main` function. Since only `Number::dec()` has a branching condition, the cyclomatic complexity of all other functions is 1. The cyclomatic complexity of `Number::dec()` is 2 because there is only one branching condition, which increases the number of independent paths through the function by one. Class `Number` has got three methods; two with a cyclomatic complexity of 1 and one with cyclomatic complexity of 2. Thus, the average cyclomatic complexity of that class is 1.33. That means, we expect the semantics of this GMSL program, applied to the given C++ program to be a function that is defined as described in Table 4.1.

<table>
<thead>
<tr>
<th>nodes(p) × names(M)</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Number :: Number, cc_per_f)</td>
<td>1</td>
</tr>
<tr>
<td>(Number :: inc, cc_per_f)</td>
<td>1</td>
</tr>
<tr>
<td>(Number :: dec, cc_per_f)</td>
<td>2</td>
</tr>
<tr>
<td>(main, cc_per_f)</td>
<td>1</td>
</tr>
<tr>
<td>(Number, avg_method_cc)</td>
<td>1.33</td>
</tr>
</tbody>
</table>

Table 4.1: Expected return function of the example GMSL program, applied to the given C++ program.

If we now apply the defined semantic functions starting with \( \mathcal{P} \), we should get a function that matches the table. We will identify the metric program with \( M \) and the given C++ program with \( p \). We get for \( \mathcal{P} \):

\[
\mathcal{P}[M]_\varsigma = C[M] (M[M]_\varsigma)_{=\mu_0}
\]

We will first examine how \( M[M]_\varsigma \) creates a metric environment \( \mu_0 \). Thereafter, we will see how \( C \) uses this environment to return the final metric.
value function.

\[
\mu_0 = \mathcal{M}[[M]](\varsigma) = \mathcal{M}'[[\text{avg}_\text{method}_\text{cc}]][\varsigma, (\mathcal{M}'[[\text{cc}_\text{per}_\text{f}]])](\varsigma, \emptyset)) = \mu_1
\]

\[
\mu_1 = \mathcal{M}'[[\text{cc}_\text{per}_\text{f}]](\varsigma, \emptyset) = \emptyset [\text{cc}_\text{per}_\text{f} \mapsto S[[\text{cc}_\text{per}_\text{f}]]](\varsigma, \emptyset, \emptyset, \emptyset)
\]

\[
\mathcal{M}'[[\text{avg}_\text{method}_\text{cc}]][\varsigma, \mu_1] = \lambda \mu_1 [\text{avg}_\text{method}_\text{cc} \mapsto S[[\text{avg}_\text{method}_\text{cc}]]](\varsigma, \mu_1, \emptyset, \emptyset)
\]

Tables 4.2 and 4.3 give an overview of the so far used metric environments.

We see that \( \mathcal{M} \) maps the names of the two metrics to functions that are determined by the semantic function \( S \), which describes the semantics of a single GMSL definition.

<table>
<thead>
<tr>
<th>name</th>
<th>( \text{nodes}(p) \to \mathbb{R} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{cc}<em>\text{per}</em>\text{f}</td>
<td>( S[[\text{cc}<em>\text{per}</em>\text{f}]](\varsigma, \emptyset, \emptyset, \emptyset) )</td>
</tr>
</tbody>
</table>

Table 4.2: Metric environment \( \mu_1 \)

<table>
<thead>
<tr>
<th>name</th>
<th>( \text{nodes}(p) \to \mathbb{R} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{avg}<em>\text{method}</em>\text{cc}</td>
<td>( S[[\text{avg}<em>\text{method}</em>\text{cc}]](\varsigma, \mu_1, \emptyset, \emptyset) )</td>
</tr>
<tr>
<td>\text{cc}<em>\text{per}</em>\text{f}</td>
<td>( S[[\text{cc}<em>\text{per}</em>\text{f}]](\varsigma, \emptyset, \emptyset, \emptyset) )</td>
</tr>
</tbody>
</table>

Table 4.3: Metric environment \( \mu_0 \)

We will first evaluate the mapping for \( \text{cc}_\text{per}_\text{f} \) and then the mapping for \( \text{avg}_\text{method}_\text{cc} \). These mappings will be functions that map \( \text{nodes}(p) \) to real numbers.

\[
S[[\text{cc}_\text{per}_\text{f}]](\varsigma, \emptyset, \emptyset, \emptyset) = S[[\text{METRIC cc}_\text{per}_\text{f} (f \ IN \ all\_funs) \ venv \ definition]](\varsigma, \emptyset, \emptyset, \emptyset) = \lambda p:\text{Prog} . \lambda n:\text{all\_funs} . S[[\text{all\_funs}]](\varsigma, \emptyset)(p) \\
\mathcal{D}[definition](\text{upd}_V(venv))(\varsigma, \emptyset) = S[[\text{cc}_\text{per}_\text{f}]](\varsigma, \emptyset, \emptyset, \emptyset)(p)
\]

The resulting lambda function is defined for all elements of \( \mathcal{G}[[\text{all\_funs}]](\varsigma, \emptyset)(p) \):

\[
\mathcal{G}[[\text{all\_funs}]](\varsigma, \emptyset)(p) = \varsigma(\text{all\_funs})(p)()
\]

For the example program \( p \), this function returns a set of nodes referring to all C++ functions of the program, i.e.

\[{\text{Number :: Number, Number :: inc, Number :: dec, main}}\]
4.1. GMSL Language

We continue with the definition part:

\[ D[\text{DEF } 1 + \text{@cn}](\varsigma, \emptyset, \eta, \nu)(p) = E[1 + \text{@cn}](\varsigma, \emptyset, \eta, \nu)(p) \]

\[ E[1 + \text{@cn}](\varsigma, \emptyset, \eta, \nu)(p) = E[1](\varsigma, \emptyset, \eta, \nu)(p) + E[\text{@cn}](\varsigma, \emptyset, \eta, \nu)(p) \]

\[ = N(1) + \nu(\text{@cn})(p) \]

\[ = 1 + B[\text{all\_cond\_nodes}(f)](\varsigma, \emptyset, \eta)(p) \]

\[ = 1 + F[\text{all\_cond\_nodes}(f)](\varsigma, \emptyset, \eta)(p) \]

As stated in the definition of \( S \), \( \eta(f)(p) \) is dependent on the current input node, which is an instance of the scope set. The possible results are listed in Table 4.4.

<table>
<thead>
<tr>
<th>( \eta(f)(p) )</th>
<th>( \varsigma(\text{all_cond_nodes})(p)(\eta(f)(p)) )</th>
<th>( \varsigma(\text{all_cond_nodes})(p)(\eta(f)(p)) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{Number :: Number}</td>
<td>\emptyset</td>
<td>0</td>
</tr>
<tr>
<td>\text{Number :: inc}</td>
<td>\emptyset</td>
<td>0</td>
</tr>
<tr>
<td>\text{Number :: dec}</td>
<td>{\text{if}(n &lt; 0)}</td>
<td>1</td>
</tr>
<tr>
<td>\text{main}</td>
<td>\emptyset</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.4: Possible results of the library function \text{all\_cond\_nodes} for program \( p \).

Integrating these results into the above equation gives us the final function that \text{cc\_per\_f} maps to in \( \mu_1 \). It associates the value 2 with the node \text{Number :: dec} and 1 with the other three possible nodes.

The evaluation of \( M'[\text{avg\_method\_cc}](\varsigma, \mu_1) \) occurs analogously. I will only go into the differences. The binding of \text{@s} is an aggregation that uses another metric. We get for the semantics of the expression \text{@s}:

\[ E[\text{@s}](\varsigma, \mu_1, \eta, \nu)(p) \]

\[ = B[\text{SUM cc\_per\_f}(m) \ OVER m \ IN \text{methods\_of\_class}(c)](\varsigma, \mu_1, \eta)(p) \]

\[ = \sum_{n \in G[\text{methods\_of\_class}(c)]}(\varsigma, \eta)(p) \]

\[ = \sum_{n \in G[\text{methods\_of\_class}(c)]}(\varsigma, \eta)(p) \mu_1((\text{cc\_per\_f})(p)(\eta[m \mapsto n](m)(p))) \]
4.1. GMSL Language

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\mu_1(cc_per_f)(p)(\eta<a href="m">m \mapsto n</a>(p))$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number :: Number</td>
<td>1</td>
</tr>
<tr>
<td>Number :: inc</td>
<td>1</td>
</tr>
<tr>
<td>Number :: dec</td>
<td>2</td>
</tr>
<tr>
<td>$\Sigma$</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 4.5: Binding value of counting variable $@s$

In this case the metric uses the metric environment $\mu_1$ to access the values of another metric. Table 4.5 lists the possible values and the summation of them.

With the result of 3 for the other counting variable $@n$ we get a new entry for the metric environment $\mu_0$ where $avg\_method\_cc$ is mapped to a function that maps the class Number to the value 1.33.

After compiling these two metrics into the metric environment $\mu_0$, the computation function $C$ uses the environment only to shift parameters in a way that the resulting function is defined to map from a pair $(n, m)$ to a real number:

$$C[M]\mu_0 = \bigcup_{m \in M} C[m]\mu_0$$
=$C[cc\_per\_f]\mu_0 \cup C[avg\_method\_cc]\mu_0$
=$\{(n, avg\_method\_cc) \mapsto \mu_0(avg\_method\_cc)(p)(n)\} \cup$
{\{(n, cc\_per\_f) \mapsto \mu_0(cc\_per\_f)(p)(n)\}}

We see that the defined semantic functions return exactly the metric value function we expected in Table 4.1.

Another important thing to show for the defined semantics is if the semantics is well-defined. In this case it is obvious that the semantics is well-defined. Firstly, GMSL does not allow loops or recursion, so that there are no infinite descending chains in the derivation of the semantic functions and secondly all types of the semantic functions are unique.

To recap the former, we can keep the image that the semantics of a metric definition is a function that maps certain scope instances to metric values. In the following we will use

$$f_m : \Pi p :Prog\_nodes(p) \rightarrow \mathbb{R}$$

as the semantic function of a metric definition $m$. 
4.2 GMSL in Goanna

GMSL is designed as an independent and portable language to define software metrics in static program analysis. As part of this work, I also developed an implementation of GMSL for the static program analysis tool GOANNA. The following sections describe the implementation and also show its correctness according to the defined semantics.

4.2.1 Metric Definitions in Goanna

Besides the definition of a metric in GMSL, GOANNA enables the user to provide some additional information of the metric. These additional information are not part of GMSL itself for different reasons. In most cases the reason is that the information just support the concrete GOANNA metric module implementation, especially the visualization, described in Chapter 6.

The entire definition of a metric in GOANNA is separated into different files, which are stated below:

<metric_name>.gmsl This file contains the actual metric definition, written in GMSL, as described in the former Section 4.1. Note that the identifier after the keyword METRIC in the GMSL definition has to be the same as the name of this file.

<metric_name>.description This file contains a string, which represents a pretty name for the metric and does not have to fit the restrictions of an identifier. It is used in the visualization as the metric’s name. If this file does not exist, or is empty, the name of the metric will be its identifier.

<metric_name>.groups This file contains a comma separated list of group names, which tag the scope of the metric. The visualization groups scope instances of computed metrics that have a common group-name. An examples for a groupname of a metric could be its scope type, like classes or functions. If this file does not exist, or is empty, the group of the metric will be set to ‘others’.

<metric_name>.mapping This file contains mapping information that map the metric values to a finite set of strings, so that you can tag special values or ranges. These mapping information will be shown in various visualizations. The mapping file also provides, to some extent, an exception handling. Values that are out of range of the mapping
will be stored in the database as ‘undefined’ and not considered in the visualization. You can define mappings by stating lines with the syntax of the following grammar:

\[
\text{mapping} = (\text{n} - \text{n}) | (\text{uc n}) \rightarrow \text{string} \\
\text{n} = [\text{0} - \text{9}] | [\text{0} - \text{9}] \cdot [\text{0} - \text{9}] \\
\text{uc} = = | < | > \\
\text{string} = ([\text{a} - \text{z}] | [\text{A} - \text{Z}] | [\text{0} - \text{9}]) +
\]

An example for a mapping file of the metric \textit{cyclomatic complexity} (see Section 4.1.1) is the following:

\[
\begin{align*}
1 : & \text{No Branching} \\
1 - 15 : & \text{Easy} \\
15 - 30 : & \text{Hard to Maintain} \\
> 30 : & \text{Extremely Complex}
\end{align*}
\]

Note that the metric value 1 will be mapped to \textit{No Branching}, since the ‘=’ operator is more specific than the interval operator. In this special case, we could have also been using

\[
2 - 15 : \text{Easy}
\]

as the second line of the mapping because we know that cyclomatic complexity can only take whole-number values, but in general, the mapping has to partition a range of float values. Therefore, the ‘=’ operator takes precedence over the other operators. In case of equal precedence, the last matching mapping will be considered (e.g. value 15 is mapped to \textit{Hard to Maintain}).

\subsection*{4.2.2 Goanna’s Metric Library}

Besides the possibility to define own metrics, \textsc{Goanna} provides a number of predefined metrics, which can be applied to programs immediately. This metric library does not have the pretension to give a set of all reasonable metrics. In fact, it should only show the expressiveness of GMSL. It contains the following metrics:

\textbf{LOC of a function} The number of source code lines of a function

\textbf{LOC of a class} The number of source code lines of a class

\textbf{Cyclomatic complexity of a function} The number of linearly independent paths through a program’s source code [McC76].

\textbf{Cyclomatic complexity of a class} The sum of cyclomatic complexities of all methods of a class.
4.2. GMSL in Goanna

**Afferent coupling of a function** The number of functions that call a certain function.

**Afferent coupling of a class** The number of classes that call a certain class [iSAa].

**Efferent coupling of a function** The number of functions that are called from a certain function.

**Efferent coupling of a class** The number of classes that are called from a certain class [iSAb].

**Instability of a function** The ratio of efferent coupling to total coupling, which is the sum of efferent and afferent coupling. \( I = \frac{Ce}{Ce + Ca} \). This metric is an indicator of the function’s resilience to change [IBM].

**Instability of a class** Same as *Instability of a function* but on class level.

**Lack of cohesion in methods of a class** The single responsibility principle states that a class should not have more than one reason to change. Such a class is said to be cohesive. A high LCOM value generally pinpoints a poorly cohesive class [NDe].

**Cohesion of a class** A measure of how strongly-related and focused the various responsibilities of a class are, depending on how many methods of a class access common fields or call common other methods [BB04].

**Largest method of a class** The largest method of a class according to the lines of code.

**Mean method size of a class** The arithmetic mean of lines of code of all methods of a class.

**Number of methods of a class** The number of methods of a class.
Chapter 5

Computation of Metrics

After the definition of metrics, we also want to compute their values on given programs. Therefore, this chapter makes some general statements on the interpretation of the definition language, before introducing the actual GMSL interpreter in GOANNA. Besides describing its architecture and some specifics, there is also a section that looks into the correctness of the implementation.

5.1 GMSL Interpretation

The computer scientist Dana Scott said in 1980:

“It is not necessary for the semantics to determine an implementation, but it should provide criteria for showing that an implementation is correct.”

According to that statement, there may be many ways to interpret GMSL definitions, but you have to make sure that they fulfill the criteria, given by the semantics, defined in Chapter 4.1.2. In fact, sometimes certain circumstances like implementation restrictions or optimizations lead to different implementations of GMSL interpreters. We will see in the next section, how certain restrictions influence the GOANNA GMSL interpreter. However, every correct GMSL interpreter has to fulfill the defined GMSL semantics. That
basically determines the input as well as the output of it. The input of an
interpreter must be a set of GMSL definitions and a program to which the
definitions are applied to. And the output is a function — or a similar data-
structure — that represents the semantic function \( f_m \), given in Chapter 4.1.2
for all defined metrics. On the other hand, the way of getting these func-
tions for an input is not determined by the semantics and can differ between
concrete implementations.

### 5.2 Goanna GMSL Interpreter

The Goanna GMSL interpreter is an extension to the existing Goanna
analyser. That means that the interpreter is used the same way the normal
analyser is. In this case it means that Goanna executes the GMSL inter-
preter right after compiling a source file. We will see how this affects the
architecture and the parameters of it.

#### 5.2.1 Architecture

The Goanna analyser is used like a normal compiler, e.g. GCC. You can
pass a file to Goanna and the analyser will first compile the file and af-
terwards check properties on the file, in order to find bugs. The metric
module can be switched on with a special parameter, which is passed to
Goanna. Figure 5.1 shows an overview of the overall control flow, which
is passed through during a metric computation. First, the analyser uses an
Edg [Groa] parser interface to extract the AST from the input program. As
the second step the analyser calls the GMSL parser to return a datastruc-
ture that contains the parsed GMSL definitions. These two datastructures
are used as the input of the GMSL interpreter. The interpreter applies the
semantics to the input and returns the computed semantic function in form
of a hashtable that maps nodes of the AST to their metric values. During
the computation the interpreter calls Goanna’s GXSL engine to evaluate
GXSL queries that are used in the GMSL definitions.

#### 5.2.2 Specifics

Besides the pure computation of the semantic function, Goanna’s GMSL
interpreter provides some outputs, which are used for further processing of
the results. To separate the actual generic interpreter from Goanna specific
customizations, the Template Method Pattern is used in GOANNA’s GMSL interpreter. The Template Method Pattern, introduced by Gamma et al. [GHJV95], abstracts away the details from several algorithms. The generic part of an algorithm is contained within a base class. Particular implementation details are captured within inherited classes. This pattern allows to separate generic GMSL interpretation parts from parts that are customized for GOANNA’s purpose. This separation is also supported by the control structure of the pattern, which is an instance of the inversion of control principle [Mar96]. Figure 5.2 shows the pattern as UML class diagramm. All generic parts of the GMSL interpreter, like the application of the semantics, are implemented in an abstract class GMSLInterpreter. This class provides some extension points, where methods that may need some customization can be implemented by other concrete classes. The GoannaGMSLAdapter class implements these extension points to match the GOANNA specifics. The output method for example is not specified in the abstract class because the output is highly dependent on the environment where this interpreter is applied. In case of GOANNA it means that the computed values are written to an SQLITE3 database. This separation of concerns also provides a single location that builds an interface between the GMSL interpreter and the GOANNA analyser. If parts of GOANNA change (e.g. changing the database), only the GoannaGMSLAdapter class have to be adjusted. The actual interpreter remains unchanged.
Another issue that the Goanna interpreter has to deal with is the application on more than one file. While the semantics definition handles the input program as a given complete AST, in practice, this AST is partially constructed and later linked in a build process. As mentioned in the introduction of Goanna’s design in Chapter 3.2.2, Goanna is used like a compiler. That means that during a build process, Goanna is called everytime a file is compiled. As a consequence of that, the GMSL interpreter has never access to the entire AST of the whole program. Besides this restriction, a computation on an AST of a whole program would cause serious performance issues, since the AST can get very big for large programs. For this reason alone, it is a good idea to implement the interpreter in a way that metrics can be computed partially on separate files.

That brings up the following questions, to be answered for an implementation:

1. How to compute metric values for scope instances that are distributed over files? The source code of scope instances (e.g., a class) can be spread over several files. If you want to compute the number of methods of a class, by querying the AST for function declarations of that class,
you will never get the number of all methods because the declarations are distributed over the ASTs of different files.

(2) *How to refer to results that already have been computed in a former file?* Assume, you have to sum the lines of code of all methods of a class. The lines of code of every single file can be computed separately, but if you want to sum them up in the end, you have to refer to results of all files that contain a method of the class.

(3) *When does the build process end?* Due to the fact that Goanna is only executed if a file is getting compiled (not even when files are linked), it is hard to say when the actual build process ends. But you have to know it if you want to distinguish between two build processes as different snapshots (described in the following chapter) or if you want to do some post processing on the metric values, after all files have been examined.

(4) *How to handle AST queries that rely on information of ASTs of other files?* An example of this is a query, where you want to return a function declaration of the source code if the function is called by another function. The GXSL query will search the AST for function calls of other functions to this function and return the declaration node if there is at least one call to it. But this query will only be correct if there are no function calls in the source code, located in other files. Otherwise, the query would also have to examine the ASTs of the other files.

Questions (1) and (2) are handled together in Goanna by establishing a kind of fixpoint computation on the files of a program\(^1\). The GMSL interpreter computes every metric separately per file, taken former results into consideration. This is done in two steps.

At first, everytime a GXSL query is executed on an AST, the resulting nodeset will also be stored in a database table. The next time this query is executed on another file (but with the same parameters, e.g. for the same class), the database entry will be updated with the union of the present entries and the new entries, queried from the current file. Thus, over the time, there will be a database entry for every used GXSL query and parameter that records the set of nodes that would have been returned when applying the function to the entire AST of the program. The return of a GXSL function then is the union of the nodeset that contains nodes from the current file and the nodeset that contains nodes from the database entry, which are results

\(^1\)Actually it is not a fixpoint computation, since the fixpoint operator does not apply the interpreter function to the output of the last iteration, but extending it.
of the same GXSL function, executed on former files. Secondly, the handling of the returned nodes in an aggregation have to be adjusted. Recalling the semantics of an aggregation from Chapter 4.1.2, the aggregator is applied to a set of numbers that is determined by the application of a query function to every node of a nodeset that is returned by another query function. As we saw before, results of a query function can now contain nodes that were stored in the database. That means they do not have a representation in the currently examined file. This causes a problem, when trying to apply the second query function of the aggregation to that node. But applying that function is unnecessary because the function has been applied to that node before, when visiting the file that contained this node. And since all GXSL query results are stored in the database, we just have to look up the entry of the query function with that node as a parameter. To summarize, the application of GXSL functions to nodes, returned by other GXSL functions, like it is used in an aggregation, makes it essential to differentiate between nodes that have a representation in the current file and nodes from former files, which are stored in the database. In the first case, a GXSL function on a node can be applied as usual to the AST of the current file. In the second case, a GXSL function application on a node results in a database lookup of the resultset that has been computed in a former file.

**Example**

As a simple example of this so called *multiple file mode* we again take the example C++ program from Chapter 4.1.2 but with a little variation. We now assume that the method \texttt{Number::dec} is located in an other file than the rest of the program. We will call the file, that contains \texttt{Number::dec}, \texttt{dec.cpp} and the file with the rest of the program \texttt{number.cpp}. Within the build process, \texttt{dec.cpp} will be compiled and examined first and \texttt{number.cpp} secondly, before everything will be linked together.

Let’s assume that we want to apply the following metric, which computes the average size of the methods of a class, according to its lines of code:

\begin{verbatim}
METRIC avg_method_size (c IN all_classes)
  WITH Os = SUM loc(m) OVER m IN methods_of_class(c),
  On = methods_of_class(c)
  DEF Os / On
\end{verbatim}

All used functions in this metric definition are AST query functions from the library.

As described above, GOANNA will evaluate the given metric firstly on \texttt{dec.cpp}. Since there is only one class (\texttt{Number}) with one method (\texttt{Number::dec})
in this file, the result of the metric application will be a function that maps the class Number to the value 4 because the only method of that class has got 4 lines of code. Additionally, all results of AST query functions are stored in a database table that looks like Table 5.1. The table stores the nodesets that are returned by GXSL library functions.

<table>
<thead>
<tr>
<th>function</th>
<th>args</th>
<th>nodeset</th>
</tr>
</thead>
<tbody>
<tr>
<td>all_classes</td>
<td>Number</td>
<td>{Number}</td>
</tr>
<tr>
<td>methods_of_class</td>
<td>Number::dec</td>
<td>{Number::dec}</td>
</tr>
<tr>
<td>loc</td>
<td>Number::dec</td>
<td>{void Number::dec(), {if (n&gt;0) n--; }}</td>
</tr>
</tbody>
</table>

Table 5.1: Database table function_results after examination of the first file dec.cpp.

After that, the second file number.cpp will be examined. This time, GOANNA will lookup the database table for additional results of library functions and join them with the results from the current file number.cpp. For example, the function methods_of_class(Number) will return \{Number :: Number, Number :: inc\} for the current file and additionally \{Number :: dec\} from the database. GOANNA joins these results by building a set union and passes them for further processing, for example, to apply another function loc(m) to them. When applying that function to the returned nodeset, GOANNA has to distinguish between nodes that come from the database and nodes that come from the current file. Depending on the origin of a node, GOANNA has to look up the database for the result of the function, or query the AST of the current file. In the example, GOANNA gets the results for loc(Number :: dec) from the database, whereas the results for loc(Number :: inc) and loc(Number :: Number) come from the current file. After all results are present, GOANNA updates the metric value for \text{avg\_method\_size}(Number) to \(\frac{4 + 4 + 1}{3} = 3\), as well as the database table entries as described in Table 5.2.

This mechanism guarantees correct measures without the need of any post processing steps after all files of a program have been examined. So question (3) is only of importance to the extent that the execution of a build process of an entire program defines a concept that we will call a snapshot and which will be described in detail in the following chapter. For now it is sufficient to recognize a snapshot as the singular execution of an entire build process. Under the introduced circumstances, question (3) is hard to answer, since you will never know when the last file of a build process is being compiled. But if we modify the question slightly by not asking for the end of a build process, but for the beginning of a new one, we are able to
5.2. Goanna GMSL Interpreter

<table>
<thead>
<tr>
<th>function</th>
<th>args</th>
<th>nodeset</th>
</tr>
</thead>
<tbody>
<tr>
<td>all_classes</td>
<td>Number</td>
<td>{Number}</td>
</tr>
<tr>
<td>methods_of_class</td>
<td>Number</td>
<td>{Number::dec,Number::inc, Number::Number}</td>
</tr>
<tr>
<td>loc</td>
<td>Number::dec</td>
<td>{void Number::dec(),{, if (n&gt;0) n--;}}</td>
</tr>
<tr>
<td>loc</td>
<td>Number::inc</td>
<td>{void Number::dec(),{, n++;}}</td>
</tr>
<tr>
<td>loc</td>
<td>Number::Number</td>
<td>{Number(int number){n=number;}}</td>
</tr>
</tbody>
</table>

Table 5.2: Database table \textit{function\_results} after examination of the second file \textit{number.cpp}.

answer it. If we store the name of a file in a database table at the time of its compilation, we can determine the beginning of a new build process to be at that time, when we examine a file that is already stored in the database table. This is true, since every file is examined exactly once during a full build process.

Question (4) is probably the hardest to answer and, unfortunately, the current GOANNA implementation is unable to do so. However, the inability of correctly evaluating GXSL functions that depend on information located in the AST of another file, is an issue of the underlying GXSL engine. We already weakened the requirements of the GXSL engine from correctly evaluating queries on entire programs to correctly evaluating queries on files, with the answer to question (1) and (2), but we still need the restriction that all information, necessary for the decision, whether a node will be returned, must be within that file. For example, a GXSL query that should return all methods of a class, will provide reasonable results, even if executed on several files because the information needed for the decision, whether a node will be returned is always in the same file (in this case it is just the name of the method’s class). But a query that should return a function declaration node, if it is called by any other function, may not correctly return all nodes that fulfill this condition. Functions that are only called by functions in other files will not be returned, since the information, necessary for the decision of returning the node, are located elsewhere. So, according to the first assumption of a correctly working GXSL engine that is able to execute queries on the AST of an entire program, this lack is not an issue of the GMSL language or the interpreter.
5.2. Goanna GMSL Interpreter

5.2.3 Correctness

The Goanna implementation is written in a functional programming language, so that most of the source code is an exact transcription of the defined semantics. If we assume that the programming language correctly implements the mathematical arithmetic as well as correct function calls and case distinctions, we can agree to a correct implementation of the semantics.

However, there are two additions to an exact copy of the semantics. The first thing is that the implementation sorts the metric definitions according to a topological order. This ensures that metrics that use other metrics are computed after the metrics they use. If there are no cyclic dependencies between metrics, a topological order always exists. A cyclic dependency’s equivalent in the semantics is a lookup in the metric environment at a point that is not defined. Thus, the behaviour is not specified and Goanna correctly implements the semantics by throwing an exception.

The second addition is the computation of metrics in the multiple file mode, described in the last section. One basic assumption of the whole method is that the results of a library function can be correctly computed by consecutively adding nodes from every file of a program to a database table. But this assumption is true if we assume an AST query engine that correctly determines whether a node fulfills a certain property according to the entire program. That namely means that the resulting nodeset of a library function is a monotonically growing set. After examining all files of the program the resultset of a function is inevitably equal to the resultset of that function, applied to the entire program. As already discussed in the last section, this assumption is a basic assumption for the design of the language. The recomputation and update of metric values after an update of a used library function ensures correct metric values even in multiple file mode.
Chapter 6

Visualization of Metrics

Software metrics are supposed to help software engineers and project managers to control and steer their software development projects. Metric values can point to bottlenecks, denote sensitive program parts, or announce accumulations of complexity within a software. In the previous sections we just introduced a metric model and described how to compute values of given metrics. But in the end there is always a human being who has to evaluate and judge these figures. To support this human to assess the values adequately, to possibly take actions, the metric module should also provide a suitable preparation of the computed metric values. The following section introduces a generic visualization model. The model defines several views that can be applied to any software metric and support the user in the evaluation of the computed figures. The section additionally describes the way how these views are implemented in Goanna.

6.1 Output and Persistence of Metric Values

One of the important monitoring actions, taken on metric values, is the development over time. Observing figures over time can denote the effects of actions that have been taken, as well as showing the need for further actions. Therefore, we have to introduce the concept of a snapshot.

**Definition 6.1.1** A snapshot $S = (M, p, t)$, with $M$ being a set of GMSL

Software Metrics in Static Program Analysis — A Formal Approach and its Application
6.2 Visualization Views and Generic Charts

definitions, \( p \) being a program and \( t \) being a timestamp, is the result of computing every metric of \( M \) on \( p \) at time \( t \).

The GOANNA GMSL interpreter associates every calculated metric with a snapshot. With this association of metric values, we can group the computed values by time, respectively by snapshot.

To integrate this concept into our image of a metric definition, we can modify our semantic function \( f_m \) from Chapter 4.1.2 in the following way:

\[
f^t_m : \Pi p : \text{Prog}. \text{nodes}(p) \rightarrow \mathbb{R}
\]

is the semantic function of a metric definition \( m \) taken at time \( t \).

We also mentioned in Chapter 4.1.2 that a metric value will be computed for every member of a set that is returned by an AST query, called the metric’s scope. This means that a metric value is additionally associated with a certain scope instance, for which the value is being computed, as well as with its metric. This is also reflected by the semantic function \( f^t_m \).

To provide computed metric values persistently, GOANNA uses an SQLite3 database with a scheme that reflects the introduced associations. An overview of the database model is depicted by Figure 6.1.

![Database model](image)

Figure 6.1: Database model for deployment of metric values

6.2 Visualization Views and Generic Charts

As a consequence of handling user-defined metrics, the visualization has to be sufficiently flexible to be used with any such metric. In contrast to a metric
6.2. Visualization Views and Generic Charts

module that provides a predefined fixed set of metrics, you have to offer a way
to visualize metrics without having any semantic information of the metric
itself. In order to design a visualization that works with all metrics defined
by the GMSL language, we have to consider what all metrics, respectively
all computed metric values, have in common:

- Every snapshot has a set of metrics that have been computed at the
  snapshot’s timestamp.
- Every GMSL metric has got a scope, which has a finite number of
  instances.
- Every instance of a scope that is used in a metric definition, can have
  additional metrics computed for that instance.

These facts are also reflected by the database model, seen in Figure 6.1.
Given these constraints, we provide four types of generic visualization views,
which can be applied to the computed metric data. These four types will be
mapped to concrete chart types that implement these views in Goanna’s
metric module.

**Time view:** The time view is a customized sequence of program snapshots
ordered by their timestamp. Let

\[ S_n = ((M_1, p, t_1), \ldots, (M_n, p, t_n)) \quad \forall 1 \leq i \leq j \leq n.t_i \leq t_j \]

be a sequence of snapshots of a program \( p \) (see Definition 6.1.1). The
time view on this sequence is defined as

\[ S'_n = (M, p, (t_1, \ldots, t_n)) \quad M = \bigcap_{i=1}^{n} M_i \]

which transforms the sequence \( S_n \), to differ just in the timestamps of
their elements. The projection of \( S'_n \) on a single metric \( m \in M \) yields
the following function, which represents the time view of one metric:

\[ h_{m}^{S'_n} : \Pi p : Prog . nodes(p) \rightarrow \mathbb{R}^n \]

\[ h_{m}^{S'_n}(node) = (f_{m}^{a1}(node), f_{m}^{a2}(node), \ldots, f_{m}^{an}(node)) \]

It maps every instance of the scope on which \( m \) is defined to a sequence
of metric values, taken at the timestamps of the snapshot sequence.

The time view in Goanna’s metric module is implemented by a stacked
bar chart, which shows the development of a metric on a project over
time. The sequence of metric values returned by \( h_{m}^{S'_n} \) are stacked for
every instance on which \( h_{m}^{S'_n} \) is defined (see Figure 6.2).
Metric view: The metric view is the summary of metric values that are computed for the instances of the metric’s scope at a certain snapshot. Formally this means, if \( s = (M, p, t) \) is a snapshot and \( m \in M \) is a metric definition, the metric view of \( m \) at snapshot \( s \) is a function \( f^s_m : \Pi p : \text{Prog-nodes}(p) \to \mathbb{R} \) that maps the instances of the metric’s scope to their metric values. This actually is exactly the semantic function \( f^t_m \) that we defined before.

In GOANNA the metric view is implemented by a horizontal barchart that contains a bar for every element \( \text{node} \) of the preimage of \( f^t_m \) with the value of \( f^t_m(\text{node}) \) (see Figure 6.3).

Scope view: The scope view is the summary of metric values that are computed for a certain instance of a scope at a certain snapshot. If \( s = (M, p, t) \) is a snapshot and \( \text{node} \in \Pi p : \text{Prog-nodes}(p) \) is an instance of a scope, then let \( M' \subseteq M \) be the set of metrics with a scope that contains \( \text{node} \) (i.e. \( \forall m \in M'. f^t_m(\text{node}) \) is defined). The scope view of a scope instance \( \text{node} \) at a snapshot with timestamp \( t \) then is a function

\[
\begin{align*}
    f^t_{\text{node}} : & M' \to \mathbb{R} \\
    f^t_{\text{node}}(m) & = f^t_m(\text{node})
\end{align*}
\]
6.2. Visualization Views and Generic Charts

In GOANNA the scope view is implemented by a radar chart that has a radar axis for every metric in $M'$ with the value of $f_{node}(m)$. This implementation is sufficient for the scope view, but GOANNA performs an additional transformation of the scope view to improve the validity of a radar chart. The intention of the scope view is to see some characteristics of a scope instance according to several metrics. But metrics can have very different ranges on which their values are defined. In most cases the comparison of two metrics on a single scope instance by its absolute metric values is not very expressive. It is more reasonable to compare the values relatively to their domain of definition, called their scale. Unfortunately, some metrics, like cohesion (see Chapter 4.1.1) are defined on a bounded set of numbers and others, like LOC, are defined on an unbounded set of numbers. It is impossible for metric values of unbounded scales to see them relatively to their scale. They first have to be mapped to a bounded set of values. We saw in Chapter 4.2 that you can enrich GMSL definitions in GOANNA with additional mapping files that partition the scale of a metric. We can represent this partitioning by a function $p_m : \mathbb{R} \rightarrow I_m$ that maps the metric values of metric $m$ to a finite image set $I_m$. This mapping is now used to project the metric values of a scope to a value of $I_m$. 

Figure 6.3: Barchart implementation of the metric view
The last step of the transformation is to rescale the metric values to compare them on a common scale. For that we need a scaling function \( s_m : I_m \to \mathbb{R} \) with

\[
s_m(v) = \frac{v \cdot \max_{i \in M'} |I_i|}{|I_m|}
\]

that expands all metric values on the same scale. The actual scope view implementation of a scope instance \( node \) and a snapshot with timestamp \( t \) in \textsc{Goanna} is a function

\[
f_{node}' : M' \to \mathbb{R}
\]

\[
f_{node}'(m) = s_m (p_m (f_{node}^t (node)))
\]

The resulting chart is given by Figure 6.4.

![Radar chart implementation of the scope view](image)

Figure 6.4: Radar chart implementation of the scope view

**Correlation view:** The correlation view is a combination of the metric view and the scope view. It allows the user to examine how the values of a set of metrics are distributed over several scope instances. If \( M' \subseteq M \) is a set of metrics with an equal scope (i.e. \( \forall m_i, m_j \in M' : f_m(none) = v_1 \iff f_{m_j}(node) = v_2 \)), the correlation view at a snapshot with timestamp \( t \)
is a function:

\[
f : \Pi p : \text{Prog} . \text{nodes}(p) \times M' \to \mathbb{R}^n \quad n = |M'|
\]

\[
f(\text{node}, \{m_1, \ldots, m_n\}) = (f_{m_1}^t(\text{node}), \ldots, f_{m_n}^t(\text{node}))
\]

The correlation view in GOANNA is only implemented for \( n = 2 \) in the form of an X-Y-Plot that shows the tuples returned by \( f \) in a two-dimensional area (see Figure 6.5).

These four introduced visualization views are conform to the generic approach because any defined metric can be observed from every of these views. I will not prove that this set of views is complete in a way that there are no other generic views, but I assert that these four views reveal a sufficient image of the metric data.

The named chart implementations in GOANNA are embedded in a website. GOANNA already implements a small webserver to provide webpages, which are used to show the progress of GOANNA’s bug detection. The webserver is extended to also provide a webpage that contains the computed metric values. Besides the actual chart implementations, the site offers a menu to navigate through projects, snapshots, metrics, and scopes. The
6.2. Visualization Views and Generic Charts

Chart implementations are generated by *Open Flash Chart 2* [Gla], which is a JavaScript library that dynamically creates *Flash* objects from JSON objects. JSON (JavaScript Object Notation) is a lightweight data-interchange format [Ecm99]. Requesting a page of the metric visualization triggers the webserver to gather the necessary information from the database, where the computed metric values are stored. The webserver takes these information afterwards and builds certain JSON objects out of it. These objects are the input for the Open Flash Chart library to generate a Flash object, which is embedded in the visualization webpage, showing the requested metric values. Which chart, respectively view, will be presented, is automatically determined by the user’s selection of snapshot, scope, and metric.
Chapter 7

Results

To test and validate the implementation of GOANNA’s metric module, I accomplished the metric calculation on the codebase of a real software project. The only requirements for the codebase were that it has to be written in C++ and it should be object-oriented to some degree. These two requirements were mainly claimed because most of my predefined metrics aimed for properties of classes and measures of the class architecture. AUDACITY [dt], an open source audio editor, fulfilled these requirements and is, with about 90,000 lines of code, of a reasonable size. With around 70 million total downloads on sourceforge.net, it is also quite popular. The integration of GOANNA into the build process of AUDACITY was fairly simple. I just had to replace the value of the compiler variable CC in the project’s Makefile from g++ to goannac++. After that, I ran several tests to evaluate the results of the computation as well as the performance. I also tested, how a large project affects the visualization module. The tests have been performed on a fairly normal desktop PC with 4 GB RAM memory and an Intel Core 2 Quad CPU @ 2.66 Mhz.

7.1 Performance

The original build process of AUDACITY uses GCC to compile and link the source code. Using this build process, AUDACITY takes 1:10 minutes to completely built. The runtime of the metric module will be composed of this compile time — because GOANNA also compiles the code — the time to
extract the AST of the source file, the parsing of the metric definitions and
the metric computation itself. To separate the computation from the parsing
steps, I ran the build process with an empty metric definition directory. That
caused the build process to compile the source code and extract the AST out
of it to be treated in GOANNA. The application of this lasted 03:04 minutes.

To measure and profile the performance of the metric computation, we set
up six different test cases. These six test runs are due to the combinations
of using one local metric, one non-local metric, and twelve miscellaneous
metrics with the metric module in single file mode and multiple file mode.
A local metric is a metric that uses only queries that can be applied directly
to the scope instance. Metric number_of_methods (see Chapter 4.2.2) is an
example of a local metric. A non-local metric, however, uses aggregations in
its definition, causing the computation to iterate over a set of nodes. Metric
largest_method (see Chapter 4.2.2) is an example of a non-local metric in
our tests because it uses an iteration over the set of methods of a class. The
twelve miscellaneous metrics contained all kinds of metrics. The test shall
show, how a range of metrics affect the runtime. All tests will be performed
with the metric module in single file mode (sfm), as well as in multiple file
mode (mfm). As described in Chapter 5.2.2, GOANNA is applied file by file
during the build process. In single file mode, the metric module treats the
AST of a file as if it were the entire program. This test might not result in
reasonable metric values when applying it to a project with several files, but
it demonstrates the overhead of the multiple file mechanism, as described in
Chapter 5.2.2.

The runtimes of these six tests, as well as the above mentioned runtimes
for GCC, the parsing steps, and GOANNA’s bug detection are shown in Fig-
ure 7.1. By profiling the application, these figures can be explained by two
main performance sinks. On the one hand the runtime heavily depends on
the number, kind, and complexity of used GXSL functions. As shown by the
difference in runtime between the computation of a local metric and a non
local metric, the use of aggregations takes significantly longer. This is due to
the iteration over nodesets, which may result in quadratic runtime, instead
of linear in terms of node instances. On the other hand the evaluation of
GXSL queries, especially on large ASTs, took the biggest proportion on the
profiling result list.

This fact slightly changes when running GOANNA in multiple file mode
(mfm). While the runtime for one metric only increases by 15-30% in compar-
ison to the single file mode (sfm), the runtime for 12 metrics almost doubled.
This overhead can be explained by three reasons. Firstly, in multiple file
Figure 7.1: Runtimes of GOANNA in different modes on the AUDACITY code-base.

mode, all query results are cached in a database, so that every application of a query causes some additional database operations. Secondly, an aggregation in multiple file mode might be more expensive because the aggregation set is expected to be larger. This is the case, if the aggregation set takes results of former files into consideration. For example, iterating over the methods of a class have more loops in multiple file mode, than in single file mode, if there are methods of that class in prior examined files. The third reason for the runtime overhead has to do with implementation issues of GOANNA, which are basically slow string operations that are used for the communication with the database.

7.2 Validation of Results

To assure the correctness of the computed metric values, we verified the results on several small test files. We additionally ran the computation on the large codebase of AUDACITY and performed some random checks on the computed figures. There are several indications that reflect the expectations of metric values for certain files. Some of them will be mentioned here.
7.2. Validation of Results

McConnell [McC93] classifies modules that handle all I/O routines, as logical cohesive. In his system of seven cohesion classes, logical cohesion is the second worst. Audacity has got two classes, named AudioIO and FileIO, which are of that kind. The computation of Badri’s [BB04] cohesion metric results in a value of 0.28 for FileIO and 0.3 for AudioIO. Since this metric is defined for values between 0 (lowest cohesion) and 1 (highest cohesion), the values support McConnell’s argument. The highest value of cohesion of the entire project had a class called WrappedType, which can be identified as functional cohesive. According to McConnell’s classification, functional cohesion is the best category.

The correlation view of some values also revealed some obvious correlations between the metrics. As Figure 6.5 already showed, there is a linear increase of the cyclomatic complexity of a class with an increasing number of methods in the Audacity codebase. This is not very surprising because the addition of a method to a class will increase its cyclomatic complexity by at least one (see Chapter 4.2.2). Another view on the correlation of cohesion and LCOM, which indicates the lack of cohesion of methods, underlines the validity of the computed values. As you would expect, an increasing cohesion value, results in a decreasing lack of cohesion. The correlation view of these values is shown in Figure 7.2.
Figure 7.2: Correlation of metric values of cohesion and LCOM (lack of cohesion of methods) on the AUDACITY codebase.
Chapter 8

Summary

This last chapter will finally summarize the results of this work and draw a conclusion. It closes with an outlook on future work, which can be performed to follow up this work.

8.1 Conclusion

This work has shown the importance of software metrics for the controlling and steering of software projects as well as for the whole evolution of software engineering to a mature engineering discipline. This work has also stated that quality assurance should be an integrated process in the entire software development. Static program analysis provides a way to test and review software already in an early stage of the development. Thus, this work introduced a way to integrate software metrics into static program analysis.

To include software metrics into static program analysis, this work defined three tasks, which have to be considered: The definition of metrics, the computation of metrics, and the visualization of metrics.

For the definition of metrics this work introduced a new programming language called GMSL. This functional language is based on queries on the abstract syntax tree of a program. The language is independent from the programming language that the defined metrics should be applied to and enables the user to define metrics on an abstract level. However, the metric
definitions can be applied directly to a program and the metric values can be computed automatically. Additionally the language allows to use other defined metrics to compose them to more complex metric definitions. Besides the syntax and some examples, this work has also defined the semantics of the language as a denotational semantics.

As proof of concept, this work has shown the implementation of a GMSL interpreter in the static program analysis tool Goanna. It has been introduced that the semantics can be implemented correctly in the environment of Goanna.

This work introduced a generic metric visualization model that defines different views on metric values, independent from the metric itself. The implementation of these views in Goanna has been illustrated.

Finally, the implementation has been tested in a case study on a real-world open source project with about 90,000 lines of code. The results of these tests were thoroughly satisfying. Relatively long runtimes have been shown as the result of implementation dependent problems and not of the introduced concepts themselves.

8.2 Future Work

I will distinguish between future work that has to be done concerning the given implementation and future work for the general concepts introduced in this work.

The Goanna implementation still has got two major problems. The first is the runtime of the metric module, which is five to ten times higher than the runtime needed for compilation and extraction of the AST alone, already for a set of only 12 defined metrics. Additionally this runtime is highly dependent on the complexity of the metrics and the size of the program’s AST. The second problem is the use of non-local GXSL queries in multiple file mode.

As already mentioned in Chapter 7.1, most of the runtime is used for the evaluation of GXSL queries. This is deeply connected to the way these functions are evaluated in the Goanna core engine. The representation of ASTs in XML may be convenient for users to read but it also leads to a very large representation. The time that is needed to evaluate query functions on such large data is unreasonably high. Similar observations can be made on the use of other libraries within Goanna, e.g. expensive string operations.
8.2. Future Work

The use of more suitable libraries and data representations will considerably decrease the runtime.

Another performance improvement might be the sharing of GXSL query results between different metrics. As already seen in the examples of Chapter 4.1.1, there are some queries that are used in several metrics, for example `all_classes`. At the moment the GOANNA implementation handles every metric independent from the others. That means a GXSL query will be evaluated once for every metric in which it is used. There will be a performance improvement if the results of such queries will be cached and reused by other metrics.

The second issue of the implementation is the use of non-local GXSL queries, i.e. queries that need information contained in other files to decide whether a node fulfills the demanded property. As already discussed in Chapter 5.2.2, the present GXSL engine is only able to correctly evaluate such non-local queries in some cases. An example where the engine fails is given in the named chapter. The adjustment of the engine to evaluate queries correctly also distributed over several files, is not only important for the metric module but will also be a requirement for an inter-procedural analysis on which the team is working right now.

Besides these two major issues of the implementation, there are a few further things that could be improved, before finally integrating the metric module into the GOANNA core product. The visualization module for example is at the moment not yet suitable for very large projects. The illustration of classes and metrics might go beyond the scope of it and the response time of the website increases distinctly. There might also be a need for an exporting function of the metric values.

Future work on the general concepts might be an extension of the GMSL language. It might be helpful in the future to also allow recursive metric definitions. This will cause little change in the syntax but major changes in the definition of semantics because recursive definitions demand fixpoint computations.

Another point which is not yet considered but can easily be integrated, is the possibility to also state GMSL expressions as anonymous metrics. The semantics can easily be extended with these *top-level expressions* by evaluating them with an empty metric environment.
Bibliography


