Software Metrics in Static Program Analysis

Andreas Vogelsang[1], Ansgar Fehnker[2], Ralf Huuck[2], and Wolfgang Reif[3]

[1] Fakultät für Informatik, Technische Universität München
Boltzmannstr. 3, 85748 Garching b. München, Germany
andreas.vogelsang@in.tum.de
[2] National ICT Australia Ltd. (NICTA)* and University of New South Wales
Locked Bag 6016, Sydney NSW 1466, Australia
{ansgar.fehnker,ralf.huuck}@nicta.com.au
[3] Lehrstuhl für Softwaretechnik und Programmiersprachen, Universität Augsburg
Universtitätsstrasse 14, 86135 Augsburg, Germany
reif@informatik.uni-augsburg.de

Abstract Software metrics play an important role in the management of professional software projects. Metrics are used, e.g., to track development progress, to measure restructuring impact and to estimate code quality. They are most beneficial if they can be computed continuously at development time. This work presents a framework and an implementation for integrating metric computations into static program analysis. The contributions are a language and formal semantics for user-definable metrics, an implementation and integration in the existing static analysis tool Goanna, and a user-definable visualization approach to display metrics results. Moreover, we report our experiences on a case study of a popular open source code base.

Keywords: software metrics, static program analysis, software quality, software maintenance

1 Introduction

Many experts from academia as well as from industry would agree on the fact that most of today's software products and their development process are of comparatively low quality. The *2009 Standish Group CHAOS Report* [15] for example states that 24% of all software projects fail, which means they are cancelled prior to completion or delivered and never used, while only 32% can be considered as successful. One of the contributing factors is that modern software is almost never completely developed from scratch, but is rather extended and modified using existing code and often includes third party source code. This can lead to poor overall maintainability, difficult extensibility and high complexity. To better understand the impact of code changes and track complexity issues as well as code quality software metrics are frequently used in the software development life cycle.

* Funded through the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council.

Ideally, software metrics should be computed continuously during the development process to enable the best possible tracking. Moreover, software metrics should be definable by development teams to not only cover general factors, but to measure company, project or team specific goals. In this work we present an integrated and flexible approach to metric computation by embedding it into static program analysis. As such, metrics can be computed on demand for every compilation even long before the software is fully developed.

In particular, we present a novel metric specification language (GMSL) that enables software developers to quickly specify their own metrics. We further define the formal syntax and semantics for GMSL, and implemented an interpreter that embeds the metric calculation in our existing static analyzer Goanna. On top of this we present a generic and user-definable visualization approach that enables quick tracking of metric results. Moreover, we report on our experiences integrating a metric specification language into static program analysis as well as our experiences from real world case studies.

Related to our approach are a number of tools that enable to compute metrics or query code for programming constructs. ODASA[4] is a commercial software assets analyzer that adds all software artefact's into a repository and provides a query engine to search for bottlenecks or quality flaws. Coverity Architecture Analysis[5] is a commercial static program analyzer for C/C++ and Java programs. It offers an architecture analysis and comes with predefined metrics that focus on complexity. Klocwork Insight[6] is another commercial source code analysis suite that includes an *Integration Build Reporting and Metrics* module for a large number of predefined metrics. NDepend[7] is a Visual Studio tool that helps the user to manage complex .NET code bases. NDepend considers the code as a database and the user can query the database and display the query results. SonarJ[8] is another software architecture management tool based on static analysis. Its main focus is to assure the consistency of the logical architecture of a system and its actual implementation. Additionally, SonarJ computes metrics, such as Robert Martin's metrics [10], and provides a histogram chart to visualize the development over time.

All of the mentioned tools can be partitioned into two different categories: Either offering a query language that allows the user to query his code for particular constructs or computing metric values on the source code during the build process based on pre-defined settings. None of the tools provide a mechanism that allows the user to define his or her own metrics that are subsequently computed automatically by the analysis tool in each compilation or build. Also, the visualizations are usually specific to the predefined metrics and measures. In contrast, our approach enables to link user-defined metrics to generic visualizations, which are independent from the metric's semantics.

[4] `http://semmle.com/technology/how-it-works/`
[5] `http://www.coverity.com/products/architecture-analysis.html`
[6] `http://www.klocwork.com/products/insight/`
[7] `http://www.ndepend.com/Metrics.aspx`
[8] `http://www.hello2morrow.com/products/sonarj/`

The next section introduces software quality metrics and static analysis, especially Goanna. Section 3, and 4 cover the metric specification language GMSL, metric computation in Goanna, and metric visualization. Section 5 discusses application of the tool to the Audacity code base, and its performance, while Section 6 concludes with an outlook on future work.

2 Integrating Software Metrics

Software metrics. Software metrics measure properties of software and are loosely defined in the IEEE 1061 standard [9] 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. "

This means that metrics make a statement about some quality attributes, are quantitative, but will have to be interpreted by a human. In this work we focus on so called *software product metrics*, which covers the aspects of *size*, *complexity*, and *quality* that can be measured on the source code and its evolution over time. Example product metrics are *lines of code*, *cohesion*, *coupling* or *cyclomatic complexity*. We will go into detail in Section 3.

While there has been a substantial body of work on metrics definitions and their correlation with program faults [7,13,14] or maintainability and bugs [4,5] we will not discuss which metrics are reasonable or particularly important. Neither will we address which metric values indicate good or poor quality. Instead we are proposing a framework that allows to define all these metrics in a flexible and concise manner and integrate them into the standard compilation and source code analysis process.

Level of Abstraction. Metrics can be defined on various levels of abstraction. Common metrics such as McCabe's cyclomatic complexity [11] are defined on the *control flow graph (CFG)* of a program and can be stated as

$$CC = e - n + 2p, \tag{1}$$

where e is the number of edges, n is the number of nodes and p is the number of strongly connected components in the CFG. Implementations, however, are typically more language specific. The tool NDepend for example defines cyclomatic complexity as:

$$CC = 1 + \{\text{number of the following expressions found in a method}\} : \\ \texttt{if|while|for|foreach|case|default|continue|goto|\&\&|||catch|?:|??} \tag{2}$$

This definition enumerates the concrete code constructs that contribute to cyclomatic complexity. These differ from language to language and the above definition is only valid for the programming language C#.

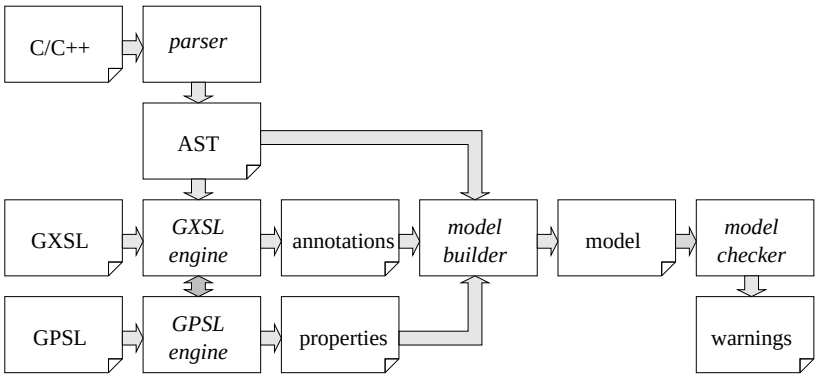

Figure 1. GOANNA's model checking approach for statically analyzing C/C++ code.

This work introduces an approach to define metrics on a more abstract level such as in (1). This means, the definition is closely related to its mathematical representation. This improves readability and maintainability of the metric definition itself. However, we also provide means to associate these definitions with elements in the *abstract syntax tree (AST)*, such that the metric definitions can be automatically computed for real-life source code.

Integrating Metric Computation. Metrics can be computed on their own or integrated into the compiler or existing source code analysis frameworks. Integration into existing frameworks leverages existing technology and requires fewer process changes for software development teams. This means, metric results are an added feature of tools that are already in frequent use.

In this work we integrate user-definable metrics in our static source code analyzer GOANNA. This tool performs deep analysis of C/C++ source code using model-checking [3] technology. GOANNA checks for bugs, memory leaks and security vulnerabilities, is fully path-sensitive and inter-procedural, and makes use of additional techniques such as abstract interpretation. A more detailed overview can be found in [6].

GOANNA already provides two specification languages for defining source code checks. The first language is a tree-query language based on XPath [2] for finding constructs and patterns of interest in the AST and is called *Goanna XPath Specification Language (GXSL)*. The second language is based on temporal logic expressions over paths in the CFG and is called *Goanna Property Specification Language (GPSL)*. GPSL allows the embedding of GXSL expression. An example is to query for `malloc` and `free` constructs in GXSL and then use the information to define in GPSL that all paths in the program from a `malloc` should lead to a `free`. Figure 1 shows how these languages feed into the static analysis. More details can be found in [16].

This work uses the existing framework and introduces a metric specification language that can reference to earlier query results, count, and compute metrics

based on arithmetic expressions. The new language will be introduced in the next section.

3 Metric Specification Language GMSL

The *Goanna Metric Specification Language (GMSL)* provides a way to define metrics on an abstract level. A prerequisite for the use of GMSL is a query engine that returns sets of nodes of the AST for which certain syntactic properties hold. As mentioned in Section 2, Goanna provides a language GXSL language to define functions that 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 particular node (or sub-trees) in the AST. The result of a GXSL query is a set of AST nodes.

Most metrics are defined for a given scope, this means for a particular set of nodes in the AST. For example, a metric might be defined for the scope *all_classes*, which means that one metric value will be computed for each class. And each class in the programm corresponds to a sub-tree in the AST. Other metrics are defined for scopes like *functions* or *namespaces*. In GMSL the scope of a metric is mention in its definition, and metric values will be computed for every instance of the scope.

GMSL distinguishes between two types of variables. One ranges over nodes (or sub-trees) of the AST, and the values are obtained by GXSL queries on the AST or sub-trees of it. These variables will be passed as arguments to other GSXL queries. The other type of variable represents integer and real numbers, which either represent the cardinality of sets, results obtained from other metrics, the result of an arithmetic expression, or the aggregated result of those. For simplicity we assume that these numbers are reals. The actual definition of the metric then is a mathematical expression containing variables over the reals, queries and constants.

3.1 Syntax

The grammar of GMSL, given in *Extended Backus Naur Form (EBNF)*, is defined in Table 1. Before we introduce the semantics, we first provide a few examples for common metrics to illustrate the language. A few functions are used in these examples, which are provided by Goanna's AST query library. This library can be extended by user-defined AST queries, e.g. GXSL functions, defined specifically to compute metrics. The following examples also demonstrate how to define a wide variety of metrics found in literature.

Cyclomatic Complexity Cyclomatic Complexity of a function as defined in NDepend is the number of branches in the control flow of a function plus one. If we only consider one function, i.e. one strongly connected component of the corresponding CFG, this definition is equal to McCabe's definition [11], which

```
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' | '_')+ ;
```

Table 1. GMSL Grammar in EBNF

defines the cyclomatic complexity as the number of linearly independent paths in the control flow of a function:

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

The metric will be computed for all nodes f returned by the GXSL query *all_funs*. It is defined as:

```
fun all_funs()
    <<./FunDecl>>
```

This function 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 the GXSL query *all_cond_nodes*, plus one. The query *all_cond_nodes* lists all conditional nodes, similar to definition (2), for C/C++:

```
fun all_cond_nodes(f)
  f<< .//If | .//While | .//For | .//Goto | .//Label | .//Default |
      .//Op2[@op='LogicalOr' or @op='LogicalAnd'] | .//Handler |
      .//Op3[@op='Cond']>>
```

Afferent Coupling Afferent Coupling of a class as defined by ARiSA[9] is the number of classes that call a certain class:

[9] http://www.arisa.se/compendium/node104.html

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

The metric will be computed for all nodes c returned by the AST query *all_classes*. The metric value of c is determined by the sum of *dependency*(g, c), applied to all nodes g, returned by the AST query *all_classes*. The AST query *dependency*(g, c) returns one node for class g, if there is a function call in class g to class c.

Cohesion Cohesion of a class as defined in [1] is a measure of how strongly-related and focused the various tasks of a class are, depending on how many methods of a class access common fields or call common other methods of the same class:

```
METRIC cohesion (c IN all_classes)
WITH @N = methods_of_class(c),
     @E = SUM directly_related(m) OVER m IN methods_of_class(c)
DEF @E /(@N * (@N-1))
```

The metric will be computed for all nodes c returned by the AST query *all_classes*. The AST query *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.

3.2 Semantics

The semantics of GMSL will be given as a denotational semantics which uses *environments* to map syntax to semantics. There are four types of environments:

- $\varsigma \in \mathit{GXSLLib}$ is a GXSL environment which maps GXSL function names to the actual GXSL functions.
- $\mu \in \mathit{MEnv}$ is a metric environment that maps metric names to their semantic function.
- $\eta \in \mathit{NEnv}$ is a node environment which maps node variables to their corresponding AST node.
- $\nu \in \mathit{VENV}$ is a variable environment which maps counting variables to their semantic value.

These environments and their product, which is denoted by *Env* are used to define the semantics of GMSL.

The semantics are defined via a function $\mathcal{M}$, which compiles a metric definition to a metric environment. All information that are necessary for applying a metric definition to a program are contained in that metric environment.

$$\mathcal{M}[\![-]\!] : \mathit{MDecl} \to \mathit{GXSLLib} \times \mathit{MEnv} \to \mathit{MEnv} \tag{3}$$

$$\mathcal{M}[\![m]\!](\varsigma, \mu) = \mu\,[\mathit{name}(m) \mapsto \mathcal{S}[\![m]\!](\varsigma, \mu, \emptyset, \emptyset)] \tag{4}$$

Function $\mathcal{S}$ maps, given an initial environment, the environment to a function that takes a program and maps the nodes of this program that are within the scope of the metric to real numbers. It is defined as follows.

$$\mathcal{S}[\![-]\!] : \mathit{MDecl} \to (\mathit{Env} \to (\Pi p : \mathit{Prog} \,.\, \mathit{nodes}(p) \rightharpoonup \mathbb{R})) \tag{5}$$
$$\mathcal{S}[\![\text{METRIC } \mathit{name}\ (\mathit{scope}\ \text{IN}\ f)\ \mathit{venv}\ \mathit{definition}]\!](\varsigma, \mu, \eta, \nu) = \tag{6}$$
$$\lambda p \in \mathit{Prog} \,.\, \lambda n \in \mathcal{G}[\![f]\!](\varsigma, \eta)(p) \,. \tag{7}$$
$$\mathcal{D}[\![\mathit{definition}]\!]\,(\mathit{upd}_V(\mathit{venv})(\varsigma, \mu, \eta[\mathit{scope} \mapsto n], \nu)(p))\,(p) \tag{8}$$

This definition reflects that a metric encompasses a *scope*, a *declaration* of counting variables, and an *arithmetic expression* over variables and applications of GMSL and GXSL functions. The set $\mathcal{G}[\![f]\!](\varsigma, \eta)(p)$ in (7) contains all scope instances. Function $\mathcal{G}$ is defined by the GXSL semantics, and returns for a given environment a set of AST nodes. Given the variable declaration part, upd_V in (8) updates $\nu \in \mathit{VEnv}$ such that it maps the counting variables to the semantics $\mathcal{B}$ of the associated *binding*. Function $\mathcal{D}$ associates the metric with the semantics $\mathcal{E}$ for the associated arithmetic expression. We omit the formal definition of $\mathcal{D}$, and upd_V for brevity; $\mathcal{E}$ will be defined below. The semantics of the bindings are defined as follows:

$$\mathcal{B}[\![-]\!] : \mathit{binding} \to (\mathit{GXSLLib} \times \mathit{MEnv} \times \mathit{NEnv} \to (\mathit{Prog} \to \mathbb{R})) \tag{9}$$
$$\mathcal{B}[\![f]\!](\varsigma, \mu, \eta) = \lambda p \in \mathit{Prog} \,.\, \mathcal{F}[\![f]\!](\varsigma, \mu, \eta)(p) \tag{10}$$
$$\mathcal{B}[\![\text{SUM } f \text{ OVER } \mathit{node} \text{ IN } g]\!](\varsigma, \mu, \eta) = \tag{11}$$
$$\lambda p \in \mathit{Prog} \,.\, \sum_{n \in \mathcal{G}[\![g]\!](\varsigma, \eta)(p)} \mathcal{F}[\![f]\!](\varsigma, \mu, \eta[\mathit{node} \mapsto n])(p) \tag{12}$$

The semantics of the remaining aggregators *PROD, MAX, MIN* are defined analogously. 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 node set, returned by another function. Simple functions in this case can be GXSL query functions from the library or the name of another GMSL metric. The semantics of a simple function f is determined by the semantic function $\mathcal{F}$. If f is a GXSL library function, $\mathcal{F}[\![f]\!](\varsigma, \mu, \eta)(p)$ in (10) or (12) returns the cardinality of the associated set. If f is a GMSL library function, it returns a real number representing a metric value.

$$\mathcal{F}[\![-]\!] : \mathit{function} \to (\mathit{GXSLLib} \times \mathit{MEnv} \times \mathit{NEnv} \to (\mathit{Prog} \to \mathbb{R}))$$
$$\mathcal{F}[\![\mathit{libfun}(n_1, \ldots, n_k)]\!](\varsigma, \mu, \eta) = \lambda p \in \mathit{Prog} \,.\, |\mathcal{G}[\![\mathit{libfun}(n_1, \ldots, n_k)]\!](\varsigma, \eta)(p)|$$
$$\mathcal{F}[\![\mathit{metric}(n)]\!](\varsigma, \mu, \eta) = \lambda p \in \mathit{Prog} \,.\, \mu(\mathit{metric})(p)(\eta(n)(p))$$

The arithmetic expression is the *definition* in semantic function $\mathcal{S}$. The semantics of these arithmetic expressions are defined as follows:

$$\mathcal{E}[\![-]\!] : \mathit{definition} \to (\mathit{Env} \to (\mathit{Prog} \to \mathbb{R}))$$
$$\mathcal{E}[\![@v]\!](\varsigma, \mu, \eta, \nu) = \lambda p \in \mathit{Prog} \,.\, \nu(@v)(p)$$
$$\mathcal{E}[\![n]\!](\varsigma, \mu, \eta, \nu) = \lambda p \in \mathit{Prog} \,.\, \mathcal{N}(n)$$
$$\mathcal{E}[\![\mathit{exp}_1 + \mathit{exp}_2]\!](\varsigma, \mu, \eta, \nu) =$$
$$\lambda p \in \mathit{Prog} \,.\, \mathcal{E}[\![\mathit{exp}_1]\!](\varsigma, \mu, \eta, \nu)(p) + \mathcal{E}[\![\mathit{exp}_2]\!](\varsigma, \mu, \eta, \nu)(p)$$

The semantics of the remaining mathematical operators $-, *, /$ are defined analogously. An expression in a definition can either be a counting variable, 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.

Example To illustrate the defined semantics consider the following metric definitions:

```
METRIC avg_method_cc (c IN all_classes)
WITH @s = SUM cc_per_f(m) OVER m IN methods_of_class(c),
     @n = methods_of_class(c)
DEF @s / @n
```

This metric `avg_method_cc` computes the average cyclomatic complexity of the methods of a class. The functions `all_classes` and `methods_of_class(c)` return the set of all class nodes (sub-tree), or for a given class node (sub-tree) the set of all method nodes (sub-trees). Function `cc_per_f(m)` is a call to another GMSL metric that computes the cyclomatic complexity per function. This metric was defined on page 5. We apply this metric definition to the following C++ program:

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

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 *Number* :: *dec*() has a branching condition its cyclomatic complexity is 2; the cyclomatic complexity of all other functions is 1. Class `Number` is in the set returned by the GXSL query `all_classes` (applied to the program), thus within its scope.

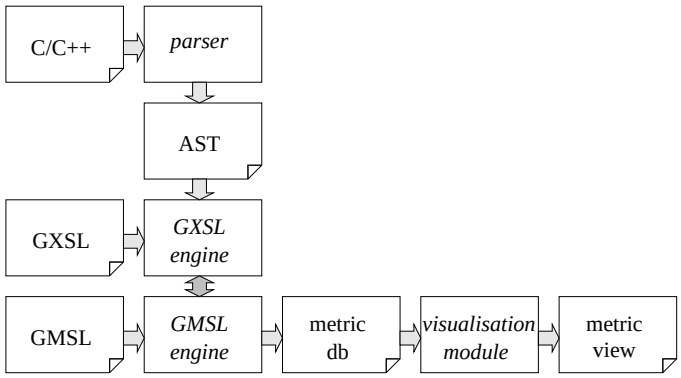

Figure 2. GOANNA's architecture for metric computation.

Variable `@s` has value $\sum_{\texttt{m} \in \mathcal{G}[\texttt{methods\_of\_class(c)}]} \mathcal{M}[\texttt{cc\_per\_f}](\texttt{m})$, i.e 4. Variable `@n` has value $|\mathcal{G}[\texttt{methods\_of\_class(c)}]|$, i.e 3 as there are three methods. Hence, the expression `@s/@n` evaluates to an average cyclomatic complexity of $1\,\frac{1}{3}$.

4 Metric Module

4.1 GMSL Interpreter

The GOANNA GMSL interpreter is an extension to the existing GOANNA analyzer. An overview of the extended architecture can be found in Figure 2. The metrics interpreter sits on top of the existing GXSL query engine, i.e., mostly uses existing library functions for pattern matching constructs of interest, and interprets the metric specification written in GMSL. Metric specifications are written in text files and that way passed to the metric module.

From an implementation point of view it is interesting to note that some metrics are incrementally computed during an analysis run with the help of a database. The reason is as follows: Some metrics require more information than what can be gathered from a local function or a single file. For instance, to compute the number of method instances of a class or computing the number of calling functions for a given callee typically requires to aggregate information from the whole project. Therefore, we use a database to store partial information where necessary and aggregate this information during the analysis of the whole program.

4.2 Visualization Module

The previous sections covered the definition and computation of metrics. However, as mentioned in Section 2 software metrics are meant to be interpreted by humans. To assist the judging process and help to understand the data we

define a generic visualization model. This enables a number of different views for a given set of metric values and allows the visualization of any user-defined metric.

To assist interpretations of the data, users of GOANNA[]'s metric module can specify information which will be used in tooltip, comments, and most importantly, to properly scale the different metrics. For the latter we implemented a user-defined mapping of GMSL output to a finite number of categories. For instance, the following ranges and categories were defined for cyclomatic complexity:

```
= 1    : No Branching
1-15   : Easy
15-30  : Hard to Maintain
> 30   : Extremely Complex
```

These categories can be used as the visualization domain for different views, and aid the interpretation of the results.

In the following we describe the four views for metric visualization implemented in GOANNA. We say $S = (M, t)$, is a *snapshot* of a project, where M is a set of GMSL metrics and t is a time stamp.

Time view: The time view is a sequence of program snapshots ordered by their time stamps. Given a sequence of snapshots $(M_0, t_0), \ldots, (M_n, t_n)$ the time view will display for each time stamp all chosen metric results per scope in M_i. This provides a good overview of how different metric values change over time. In the visualization module this will be displayed as a stacked bar chart as seen in Figure 3(a).

Metric view: The metric view is the summary of one metric for all elements in one scopes at one point in time, i.e., for a single snapshot (M, t). In GOANNA the metric view is implemented by a horizontal bar chart that lists the metric values of different elements in the scope in decreasing order. Figure 3(b) shows an example for the ranking of classes by cohesion.

Scope view: The scope view is the summary of all metric values that are computed for a certain instance of a scope at a certain time. The scope view is implemented by a radar chart where every axis represents a metric. An example for the different metric values of a given class is given in Figure 4(a).

Correlation view: The correlation view is a combination of the metric view and the scope view. It enables the user to examine how the values of a pair of metrics correlate over several scope instances. The correlation view is implemented in the form of an X-Y-Plot. See Figure 4(b) for an example.

The different metric views are configurable and can be combined in a dashboard if desired, but most importantly they are independent from a metric itself. As such they can visualize any metric and sufficiently provide a quick overview of the status of a software project.

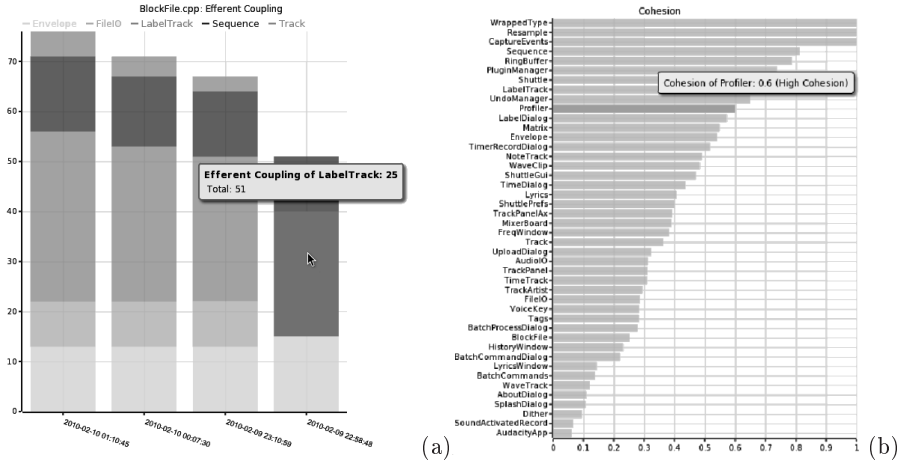

Figure 3. (a) Histogram implementation of the time view. The histogram shows the efferent coupling over time for different classes. (b)Bar chart implementation of the metric view. Ranking of classes by cohesion.

5 Case Study

This section reports on the application of Goanna's metric module to the *Audacity*[10] code base. Audacity is an open source audio editor and written in C++. The latter was essential for testing the metrics defined for classes. With about 90,000 lines of code it has a reasonable size, and is, with around 70 million total downloads on *sourceforge.net*, also quite popular. The tests were performed on a desktop PC with 4 GB RAM and an Intel Core 2 Quad CPU @ 2.66 Mhz. The results for an implementation of the metric module based on Goanna version 1.1.

The original build process of Audacity uses Gcc to compile and link the source code. This build process takes 1:10 minutes to complete. 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 files, the parsing of the metric definitions, and the metric computation itself. To separate the computation from the parsing steps, the module was run with an empty metric definition. Compiling the source code and extracting the AST took 03:04 minutes.

To measure and profile the performance of the metric computation, we set up six different test cases. These test runs are combinations of using one local metric, one non-local metric, and twelve miscellaneous metrics. Moreover, each of these cases were run in single file mode (sfm) and multiple file mode (mfm). A local metric is a metric that uses only queries that can be evaluated directly

[10] `http://audacity.sourceforge.net/`

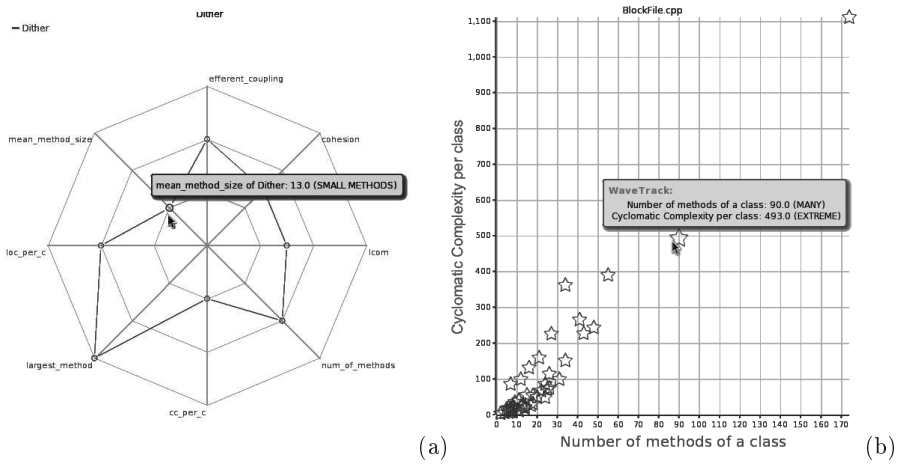

Figure 4. (a) Radar chart implementation of the scope view. All metric values for a given class. (b) X-Y-Plot for the correlation view. This figure correlates the number of methods of a class, with the cyclomatic complexity.

on the local scope instance. For instance, the metric *number_of_methods* is a local metric. A non-local metric, in contrast, iterates over sets of nodes that span multiple files. Metric *avg_method_cc* is an example, since it iterates over the set of methods of a class, which may be distributed over multiple files.

Among the twelve metric we measured were: *Cyclomatic complexity*, *Afferent coupling*, *Efferent coupling*, and *Instability* [8] of classes and functions, and *Lack of cohesion in methods of a class (LCOM)*.

The runtimes of these tests as well as the above mentioned runtimes for GCC and the GOANNA's bug detection (`goannac++`) are shown in Figure 5.

One immediate observation is that the runtimes heavily depend on the number, kind, and complexity of the GXSL functions used. 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 node sets, 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 of time.

Another observation is that when running GOANNA in multiple file mode (mfm) for one metric the runtime only increased by around 15-30% in comparison to the single file mode (sfm), the runtime for 12 metrics roughly doubled. This overhead can be explained by three reasons: Firstly, in multiple file mode all query results are stored in a database. Hence, every application of a query causes some additional database operations. Secondly, an aggregation in multiple file mode can be more expensive, because the aggregation set is typically larger. The third reason for the overhead had to do with slow string operations that were used for the communication with the database.

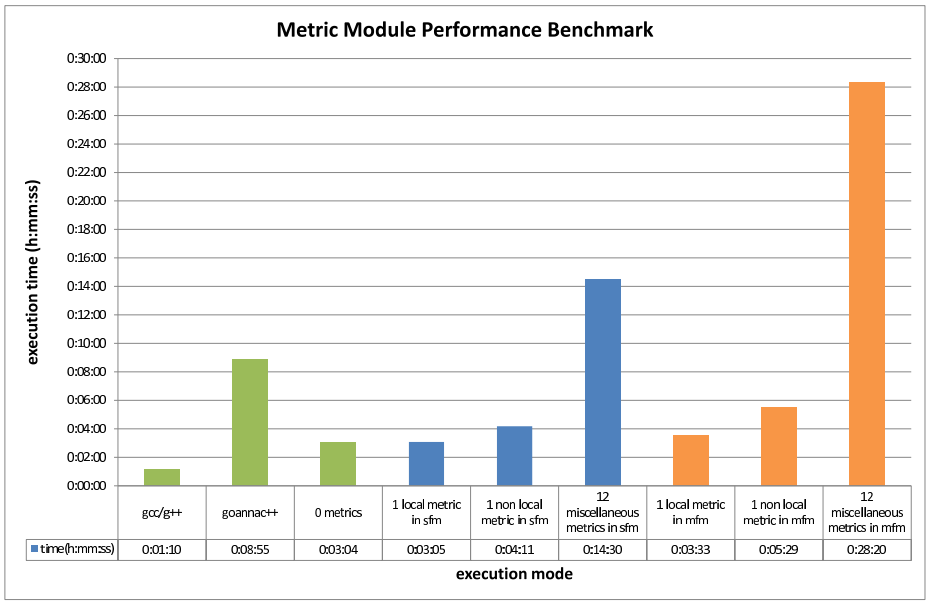

Figure 5. Runtimes of GOANNA version 1.1 in different modes on the Audacity code base.

Some of the performance issues have been addressed in later versions of GOANNA, but we like to point out that the current implementation is a prototype and has a lot of room for improvement. What is more important is that we were able to easily specify metrics and experimentally confirm some of the arguments brought forward in the literature as we see next.

Notable Results. The results we obtained were compared to some claims made by other authors. For instance, McConnell [12] 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 two I/O classes, named *AudioIO* and *FileIO*. The results obtained by the metric module confirm McConnell's conjecture: The cohesion computed by GOANNA according to Badri's [1] formula resulted in 0.28 for *FileIO* and 0.3 for *AudioIO*, which is on the low end of the spectrum. 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 expected connection between the metrics. As Figure 4(b) showed, there is a linear correlation between cyclomatic complexity of a class with an increasing number of methods in the Audacity code base. Of course, one simple contributing factor is that the addition of a method to a class will increase its cyclomatic complexity by at least one. Another observation is the correlation between *cohesion* and *LCOM*, which

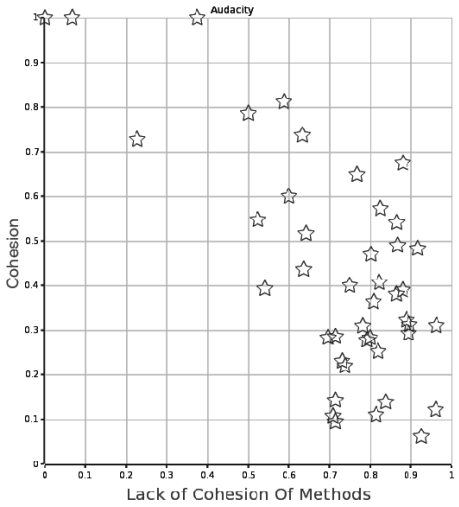

Figure 6. Correlation of metric values of *cohesion* and $LCOM$(lack of cohesion of methods) on the Audacity code base.

indicates the lack of cohesion of methods. As one might expect, an increasing cohesion value results in a decreasing lack of cohesion. The correlation view of these values for the Audacity code base is shown in Figure 6.

6 Conclusions

In this work we presented an approach to user-defined software metrics and a seamless integration into static program analysis. Unlike existing approaches the metrics are not hard coded, but interpreted at analysis time from a textual description that can be defined by software developers and teams themselves. The specification language GMSL is based on a formal syntax and semantics. While we chose to integrate the interpreter in our own tool there is in principle no restriction for using the same approach in, e.g., the standard compiler.

On top of the metric specification language we built the proof of concept of a generic metric visualization module. This module enables the mapping of any metric to different views and the automatic user-defined mapping of values to abstract categories. In practice, this has been proven useful to quickly assess the state of a software project.

Future work has to address some of the current implementation issues, such as relatively slow database access and optimizing the query interpretation. Moreover, some work has to go into scaling the used visualization techniques to large software projects. Once the user is confronted with dozens of metrics and thousands of files it is important to have some automated visual abstraction to avoid confusion and overload.

References

1. Badri, L., Badri, M.: A proposal of a new class cohesion criterion: An empirical study. Journal of Object Technology 3(4), 145–159 (2004)
2. Clark, J., DeRose, S.: XML Path Language 1.0 (XPath). W3C (1999), http://www.w3.org/TR/xpath
3. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge, MA, USA (1999)
4. Curtis, B., Sheppard, S.B., Milliman, P.: Third time charm: Stronger prediction of programmer performance by software complexity metrics. In: Proceedings of the Fourth International Conference on Software Engineering. pp. 356–360. IEEE Computer Society Press (1979)
5. Elshoff, J.: An analysis of some commercial PL/I programs. IEEE Transactions on Software Engineering SE-5(2), 113–120 (1976)
6. Fehnker, A., Huuck, R., Jayet, P., Lussenburg, M., Rauch, F.: Model Checking Software at Compile Time. In: Proceedings of the 1st International Symposium on Theoretical Aspects of Software Engineering. Shanghai, China (2007)
7. Ferzund, J., Ahsan, S.N., Wotawa, F.: Empirical evaluation of hunk metrics as bug predictors. In: Abran, A., Braungarten, R., Dumke, R.R., Cuadrado-Gallego, J.J., Brunekreef, J. (eds.) Software Process and Product Measurement, International Conferences IWSM 2009 and Mensura 2009. Lecture Notes in Computer Science, vol. 5891, pp. 242–254. Springer (2009)
8. IBM: In pursuit of code quality: Code quality for software architects. Website, http://www.ibm.com/developerworks/java/library/j-cq04256/; visited on 3 February 2010
9. IEEE: IEEE Standard for a Software Quality Metrics Methodology. Institute of Electrical and Electronics Engineers (1061)
10. Martin, R.C.: Agile software development: principles, patterns, and practices. Alan Apt series, Prentice-Hall, pub-PH:adr (2003)
11. McCabe, T.J.: A complexity measure. IEEE Transactions on Software Engineering 2(4), 308–320 (1976)
12. McConnell, S.: Code Complete: A Practical Handbook of Software Construction. Microsoft Press (1993)
13. Misra, S.C., Bhavsar, V.C.: Relationships between selected software measures and latent bug-density: Guidelines for improving quality. In: Kumar, V., Gavrilova, M.L., Tan, C.J.K., L'Ecuyer, P. (eds.) Computational Science and Its Applications - ICCSA 2003. Lecture Notes in Computer Science, vol. 2667, pp. 724–732. Springer (2003)
14. Nagappan, N., Ball, T., Zeller, A.: Mining metrics to predict component failures. In: ICSE '06: Proceedings of the 28th international conference on Software engineering. pp. 452–461. ACM, New York, NY, USA (2006)
15. The Standish Group: Chaos report 2009. Website, http://www1.standishgroup.com/newsroom/chaos_2009.php; visited on 25 February 2010
16. Vistein, M., Ortmeier, F., Reif, W., Huuck, R., Fehnker, A.: An abstract specification language for static program analysis. Electr. Notes Theor. Comput. Sci 254, 181–197 (2009)