COOPERATION

OSSMETER

Automated Measurement and Analysis of Open Source Software

Project Number 318772

D3.1 – Report on Domain Analysis of OSS Quality Attributes

Version 1.0
21 October 2013
Final

Public Distribution

Centrum Wiskunde & Informatica

Project Partners: **Centrum Wiskunde & Informatica**, **SOFTEAM**, **Tecnalia Research and Innovation**, **The Open Group**, **University of L′Aquila**, **UNINOVA**, **University of Manchester**, **University of York**, **Unparallel Innovation**

Project Partner Contact Information

Centrum Wiskunde & Informatica Paul Klint Science Park 123 1098 XG Amsterdam, Netherlands Tel: +31 20 592 4126 E-mail: paul.klint@cwi.nl	**SOFTEAM** Alessandra Bagnato Avenue Victor Hugo 21 75016 Paris, France Tel: +33 1 30 12 16 60 E-mail: alessandra.bagnato@softeam.fr
Tecnalia Research and Innovation Jason Mansell Parque Tecnologico de Bizkaia 202 48170 Zamudio, Spain Tel: +34 946 440 400 E-mail: jason.mansell@tecnalia.com	**The Open Group** Scott Hansen Avenue du Parc de Woluwe 56 1160 Brussels, Belgium Tel: +32 2 675 1136 E-mail: s.hansen@opengroup.org
University of L′Aquila Davide Di Ruscio Piazza Vincenzo Rivera 1 67100 L'Aquila, Italy Tel: +39 0862 433735 E-mail: davide.diruscio@univaq.it	**UNINOVA** Pedro Maló Campus da FCT/UNL, Monte de Caparica 2829-516 Caparica, Portugal Tel: +351 212 947883 E-mail: pmm@uninova.pt
University of Manchester Sophia Ananiadou Oxford Road Manchester M13 9PL, United Kingdom Tel: +44 161 3063098 E-mail: sophia.ananiadou@manchester.ac.uk	**University of York** Dimitris Kolovos Deramore Lane York YO10 5GH, United Kingdom Tel: +44 1904 325167 E-mail: dimitris.kolovos@york.ac.uk
Unparallel Innovation Nuno Santana Rua das Lendas Algarvias, Lote 123 8500-794 Portimão, Portugal Tel: +351 282 485052 E-mail: nuno.santana@unparallel.pt	

Contents

Document Control

Version	Status	Date
0.1	First version	26 September 2013
0.2	Second version (Changes reflecting internal review)	10 October 2013
1.0	First Public Version	21 October 2013

Executive Summary

Meaningful and effective measurement of quality attributes of Open Source Software (OSS) requires:

- Analysis of and insight in the domain of software quality measurement.
- Identification of relevant metrics to measure software quality attributes.
- A meta-model to store the results of measurement (i.e., facts directly extracted from the source) as well as any metrics that are derived from these measurements.
- Calculation of metrics based on the extracted facts.

In this deliverable we present:

- A brief summary of motivation and challenges for Task 3.1.
- Introduction to software quality as described in ISO/IEC 9126-1:2001.
- An initial set of requirements for quality attributes in OSSMETER.
- A survey of the domain of software quality.
- An initial selection of metrics that are relevant for measuring software quality attributes.
- A tool survey of existing tools for metric calculation.
- A quick overview of the RASCAL meta-programming language that will be used for metric calculation.
- A proposal for and illustration of the Metrics Meta-Model (M3), a general framework for representing basic facts as well as derived (computed) metrics. We show how M3 can be extended to represent Java-specific facts and how these facts form the basis for metric computation and visualization. The presented metrics are already useful but are primarily intended as a proof-of-concept of the approach.
- An estimation how well the general OSSMETER requirements for WP3 can be satisfied.

1 Introduction

Many different indicators of an OSS project determine its quality but the quality of its source code and the quality of its developer community are among the top ones. Regarding source code quality, elementary indicators such as Source Lines of Code (SLOC) can be used and are highly relevant. However, many variations exist even for such a simple metric (count lines with/without comments or blank lines, count statements rather than lines, etc.) and we need project-wide consistency to base OSS project comparisons on. Slightly more advanced metrics consider cohesion and coupling of components, consistency of identifiers using nano-patterns [17] and the occurrence of certain code smells [20].

We make a distinction between generic (or language-agnostic) methods that can be used to analyze across different implementation languages in an OSS project, and language-specific methods that are only applicable to one implementation language. We make a further distinction between text-based and syntax- based techniques. The latter require the creation or adaptation of grammars for the relevant languages, and the availability of appropriate tool generation techniques.

Regarding the developer community, the information source most closely tied to the source code is the version management system. This will be used to measure developer activity and involvement in the course of the project. Examples of language-agnostic developer involvement and code evolution properties are:

1. The distribution of developers over time and location.
2. The evolution over time of each component (in order to identify high-activity components).
3. Co-evolution of components (important to identify implicit component dependencies).

Examples of language-specific development activities are:

1. Correlation between evolution and code metrics.
2. Co-evolution of components that do not explicitly invoke each other.
3. Correlation between code smells and specific developers.

In order to obtain the relevant properties we need:

1. Parsers for all relevant implementation languages.
2. Fact extractors that can extract all the information from source code that is needed to compute the selected metrics.
3. Metrics calculators that synthesize the extracted facts to the required metrics.
4. Analysis tools that perform the required smell detection.
5. Reporting and visualization to summarize the findings per project.

For this we will largely depend on the Rascal meta-programming language.[1]

[1] `http://www.rascal-mpl.org`

1.1 Outline of Deliverable 3.1

A domain analysis is needed to explore the domain of OSS software quality measurement and to make an inventory of the measurement tasks that are needed to support it. The following questions have to be answered:

- Which source code and version management attributes are most relevant for tracking the quality of OSS projects?
- What are the observable effects of these attributes in the source code and the version management system?
- How can these attributes be measured? An important issue here is whether a quality metric can be measured in a generic (or language-agnostic) fashion (i.e., one measurement tool can be used for all possible software implementation languages encountered in OSS projects) or in a language-specific fashion (i.e., the quality metric has to be measured by a different tool for each different language).
- What are the costs of measurement tools for each metric (=cost of measurement implementation)?
- What are the tradeoffs between accuracy and efficiency for each metric? (= cost of measurement application)?
- What tools (existing or new) are needed to provide the required measurements.

The domain analysis will include a literature and tool survey and will offer recommendations which combinations of metrics and tools are most cost effective.

Based on the domain analysis, a metamodel will be developed for describing quality attributes for OSS projects. This metamodel will be explicitly linked to the overall OSS project metamodel developed in WP2 and the delineation between these models will have to be discussed among the partners. The software quality attribute model will include, amongst others, the following information:

- Identification information for the project that links it to the overall OSS project metamodel.
- Information about the implementation languages used in the project.
- Information about the (build, configuration, version management) tools used in the project.
- Information about the source licenses that have been found in the source code.
- Information about external tools and libraries on which the project depends.
- Language-agnostic metrics. Examples of possible candidates are SLOC per implementation language and text-based code clone metrics.
- Language-specific metrics. Examples of possible candidates are counts for classes, methods, functions, and more detailed metrics for McCabe complexity, and coupling an cohesion among modules.

The structure of this deliverable is as follows:

- Section 2 explores how to design a quality model for OSSMETER.
- Section 3 describes essential quality attributes for OSS projects.
- Section **??** describes our initial selection of metrics for measuring these quality attributes. This section also presents a brief tool survey.

- Section 4 gives a quick overview of the RASCAL meta-programming language that will be used for fact extraction and metric calculation.
- Section 5 describes M3 (Metrics Meta-Model) that we have designed for represented facts and metrics in OSSMETER. The core M3 model and an extension for Java are presented.
- Section 6 briefly illustrates the possibilities for visualizing facts and metrics that are stored in an M3 model.
- Section 7 summarizes the compliance of the solutions proposed in this deliverable with the overall OSSMETER requirements.
- Appendix A shows an example of facts extracted from a Java project and represented in the M3 Java Model.

2 Towards a Software Quality Model

The question raised in the previous section should be considered in the more global perspective of software quality as described in the ISO/IEC 9126-1:2001 standard that distinguishes the following aspects of software quality.

2.1 The ISO/IEC 9126-1:2001 quality model

Functionality A set of attributes that bear on the existence of a set of functions and their specified properties. The functions are those that satisfy stated or implied needs.

- Suitability.
- Accuracy.
- Interoperability.
- Security.
- Functional compliance.

Reliability A set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for a stated period of time.

- Maturity.
- Fault tolerance.
- Recoverability.
- Reliability compliance.

Usability A set of attributes that bear on the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users.

- Understandability.
- Learnability.
- Operability.
- Attractiveness.
- Usability compliance.

Efficiency A set of attributes that bear on the relationship between the level of performance of the software and the amount of resources used, under stated conditions.

- Time behavior.
- Resource utilization.
- Efficiency compliance.

Maintainability A set of attributes that bear on the effort needed to make specified modifications.

- Analyzability.
- Changeability.
- Stability.
- Testability.
- Maintainability compliance.

Portability A set of attributes that bear on the ability of software to be transferred from one environment to another.

- Adaptability.
- Installability.
- Co-Existence.
- Replaceability.
- Portability compliance.

2.2 Other Quality Attributes

There are, however, quality aspects of OSS systems and projects that are not covered by the ISO quality model:

- The strength of the development community.
- The quality of the development process.
- The economic support and viability.

2.3 Requirements for OSSMETER quality attributes

The attributes of the OSSMETER quality model will be guided by the following requirements and constraints[2]

- They should be directly relevant for commercial and non-commercial users of OSS.
- They can be combined in ways that reflect the needs of a specific user or user community. Hence separate metrics need to be provided both as predefined and as user-definable combinations of metrics.
- They can be measured on the basis of artifacts that are directly available and can be directly measured. This includes source code, information from development tools (version management, bug tracking), and other source (e.g., bulletin boards, e-mail lists).
- They can be measured by static analysis of the artifacts only.

We will now discuss an initial selection of software quality attributes that satisfy these requirements.

[2]In Section 7 we give an estimation how well the OSSMETER requirements can be satisfied.

3 OSS Quality Model Attributes & Metrics

3.1 Attributes Defining Quality of OSS Projects

Given the analysis and requirements just described, we propose the following initial set of quality attributes for source code and for the activities related to the development of source code[3].

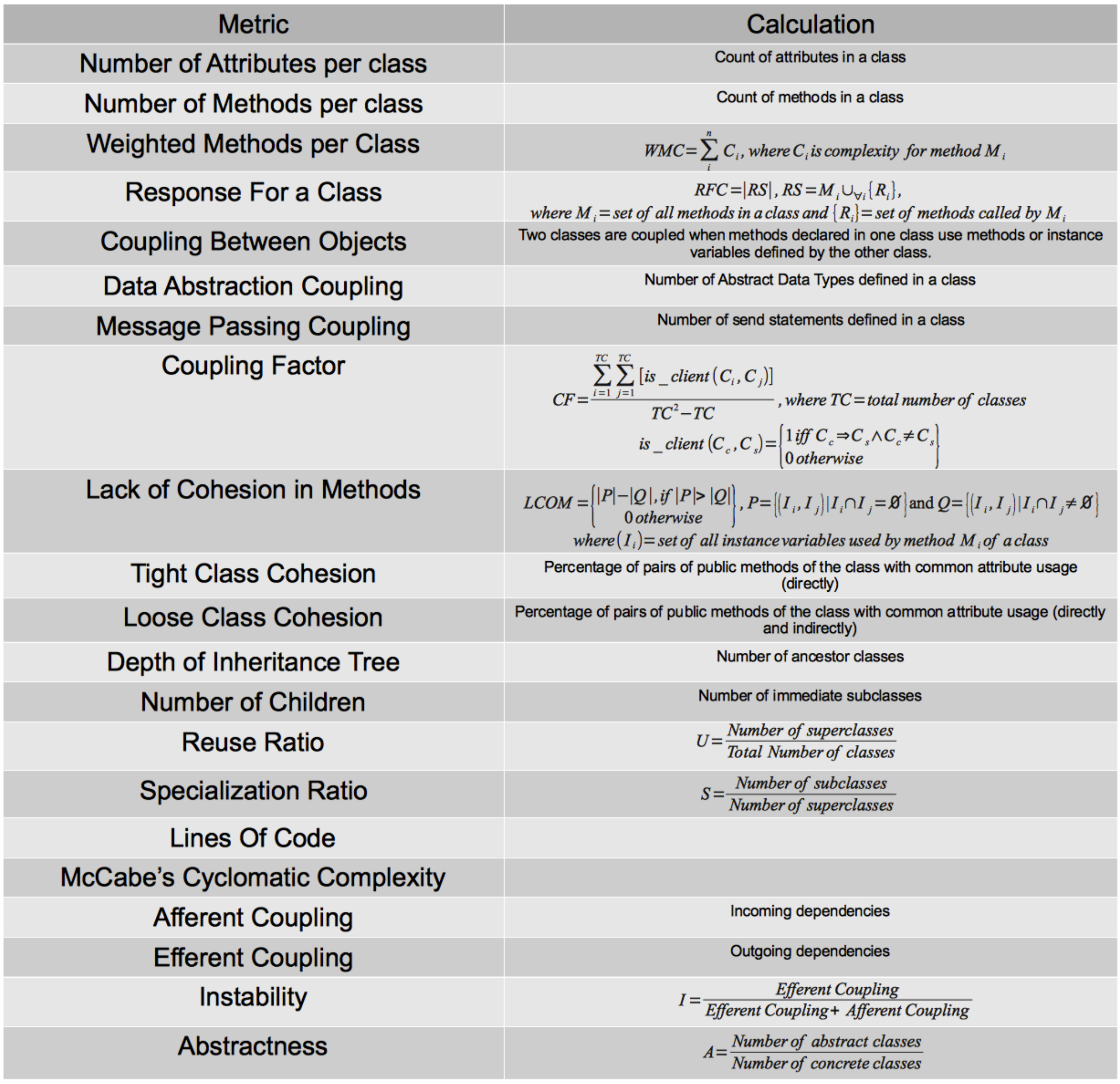
3.1.1 Source Code Attributes

The source code attributes that we have identified as being most relevant for determining quality are:

- **Size**: The larger a system becomes, the harder it becomes to understand and maintain it. Size manifests itself in lines of code, number of components and the like.
- **Complexity**: Impacts understandability and maintenance. Complexity can be caused by size, by complex control flow inside components or by complex interactions between components.
- **Documentation**: Documenting complex parts of the system helps in understanding the system better. This assumes existence of documentation.
- **Coupling**: The more *external* connections (couplings) there are between components, the harder they are to understand in isolation. A coupling can be a control-flow coupling (one component calling a method from another component) or a data coupling (one component using a datatype defined in another component).
- **Cohesion**: The more *internal* connections (couplings) there are between internal parts of a component, the more cohesive it is: these couplings explain why these internal parts belong to that component in the first place.
- **Code smells**: Certain coding techniques go against what are considered to be best practices and are called code smells. Examples are functions or methods with excessively long bodies, the use of switch statements in an OO language or the use of casts or `instanceof`.

Source code attributes are domain independent in the sense that they dictate quality for all software projects, not just open source ones. The benefit of having these attributes in OSSMETER, apart from showing source code quality, is to provide users (developers who want to contribute/extend open source projects) with an indication of how easy/hard their task will be.

[3] We would like to emphasise that this is an initial list that can be expanded as needed



Metric	Calculation
Number of Attributes per class	Count of attributes in a class
Number of Methods per class	Count of methods in a class
Weighted Methods per Class	$WMC=\sum_{i}^{n} C_i$, *where* C_i *is complexity for method* M_i
Response For a Class	$RFC=\lvert RS\rvert,\ RS=M_i \cup_{\forall i}\{R_i\},$ *where* M_i *= set of all methods in a class and* $\{R_i\}$ *= set of methods called by* M_i
Coupling Between Objects	Two classes are coupled when methods declared in one class use methods or instance variables defined by the other class.
Data Abstraction Coupling	Number of Abstract Data Types defined in a class
Message Passing Coupling	Number of send statements defined in a class
Coupling Factor	$CF=\frac{\sum_{i=1}^{TC}\sum_{j=1}^{TC}[is\_client(C_i,C_j)]}{TC^2-TC}$, *where* TC *= total number of classes*; $is\_client(C_c,C_s)=\begin{cases}1\ iff\ C_c \Rightarrow C_s \wedge C_c \neq C_s\\ 0\ otherwise\end{cases}$
Lack of Cohesion in Methods	$LCOM=\begin{cases}\lvert P\rvert-\lvert Q\rvert, if\ \lvert P\rvert>\lvert Q\rvert\\ 0\ otherwise\end{cases}, P=\{(I_i,I_j)\mid I_i \cap I_j=\emptyset\}$ and $Q=\{(I_i,I_j)\mid I_i \cap I_j \neq \emptyset\}$ *where* (I_i) *= set of all instance variables used by method* M_i *of a class*
Tight Class Cohesion	Percentage of pairs of public methods of the class with common attribute usage (directly)
Loose Class Cohesion	Percentage of pairs of public methods of the class with common attribute usage (directly and indirectly)
Depth of Inheritance Tree	Number of ancestor classes
Number of Children	Number of immediate subclasses
Reuse Ratio	$U=\frac{Number\ of\ superclasses}{Total\ Number\ of\ classes}$
Specialization Ratio	$S=\frac{Number\ of\ subclasses}{Number\ of\ superclasses}$
Lines Of Code	
McCabe's Cyclomatic Complexity	
Afferent Coupling	Incoming dependencies
Efferent Coupling	Outgoing dependencies
Instability	$I=\frac{Efferent\ Coupling}{Efferent\ Coupling+Afferent\ Coupling}$
Abstractness	$A=\frac{Number\ of\ abstract\ classes}{Number\ of\ concrete\ classes}$

Table 1: Common metrics

Quality attribute	Related metrics
Size	• Lines of code (LOC) 1. LOC (non-empty) 2. LOC (non-empty, non-commented) • Number of methods per class • Number of attributes per class • Abstractness • Reuse ratio • Specialization ratio • Depth of inheritance tree • Number of children
Complexity	• Cyclomatic complexity • Weighted methods per class • Response for a class
Documentation	• Density
Coupling	• Coupling between objects • Data abstraction coupling • Message passing coupling • Coupling factor • Afferent coupling • Efferent coupling • Instability
Cohesion	• Lack of cohesion in methods (many variants available) • Tight class cohesion • Loose class cohesion
Contibutors	• Number of contributors • Number of new contributors (in a time frame) • Number of inactive contributors
Rate of change	• Number of commits (in a time frame)
Core contributors	• Contributors with a large number of commits compared to other commiters • Contributors who are largely responsible for components in the project

Table 2: Quality attributes and metrics to calculate them.

3.1.2 Activity Attributes

Activity attributes on the other hand play a much more significant role in the open source domain. Knowing how fast bug fixes are made, how the project is changing over time or even knowing that the number of contributors has been declining will provider users means to decide if they wish to adopt a certain project. Some activity attributes could include:

- **Contributors**: Information on contributors like the number of contributors can be an indicator of the health of an open source project. Constant increase in contributors who make frequent commits could be an indicator that interest in the project is growing.
- **Core contributors**: Identifying core contributors for a project would give us possibilities to generate early warnings in case the contributor is no longer actively involved in the project. Loss of a contributor usually means loss of some knowledge which depending on the situation could lead to a big maintenance issue for the rest of the contributors.
- **Rate of change**: Rate of change could provide an indication on how active the development community is.
- **Bug Fixes**: The time between a bug report and fix could be an indicator for how actively a system is maintained by the development community.
- **Evolution over time**: Evolution of the quality attributes over time would provide indicators for the health of the project.
- **Co-evolution**: Identifying co-evolving components would provide for better understanding of the effects that might propagate through a system when changing a component.

3.2 Common metrics for measuring quality attributes

In Table 1 we summarize the definitions of some common metrics that we will discuss now. The main sources for these definitions are [13], [6], [2], [12], [4].

In Table 2 we show which metrics can be used to calculate the quality attributes defined in section 3. As mentioned in footnote 3, this is an initial list that is open for extension.

All the metrics chosen in the initial list are well known metrics in the object-oriented paradigm. The decision to start with these metrics, lies in the fact that some of the most popular languages for open source projects fall under this paradigm (like Java, C++, PHP). The goal is to start with analysis of Java projects and slowly integrate new languages. In time, we will also integrate languages from other paradigms, again based on their popularity in the open source community.

3.3 Presentation of Metrics

Metrics like lines of code (LOC), defect counts, cyclomatic complexity have been historically used to measure the quality of projects and productivity. Research has shown that metrics by themselves have not been a good indicator/predictor for quality or productivity[5]. This leads us to question how should we present the computer metrics for the various quality attributes, to the users of OSSMETER. Key requirements here are transparency and adaptability.

There are two complementary approaches. First, to present the raw metrics calculated for a system to the users so that they are free to interpret these values subjectively (since no one model will be to the liking of everyone). Second, to provide a number of quality models to give users more options in determining the quality of a system. by selecting model parameter specific for their needs. Examples are the SIG Maintainability Model [7] and the SQO-OSS Quality Model [15]. This allows the users to use models that they are comfortable with (if any). In addition, this allows us to add or remove models based on further advancements in the field.

The evaluation of quality observations should mostly be left up to the user to decide. For some metrics, we can provide evaluations based on quality models as just described. For other metrics, we allow users to set the threshold values for the metrics that they would like to set for different quality levels of a system. Typically, we could have 5 scales for quality:

1. Very Good
2. Good
3. Fair
4. Poor
5. Very Poor

We present the calculated metric values for each system and present them to the users in a concise manner. It is then for the user to decide the threshold values for each metric to separate them into the different scales.

3.4 Tool Survey

A tool survey was done to identify existing tools that could be used for metric calculation for OSSMETER. Research has shown that metric values calculated by different tools may vary. The implementation is dependent on the interpretation of the creators of the tool. As such considering the aims of OSSMETER we used two primary criteria for tools selection.

1. **Metrics Supported**

   The tool should support a variety of metrics (for example, lines of code, object oriented metrics suite like C&K metrics suite, complexity, duplication detection).

2. **Languages Supported**

   The tools should support as many languages as possibles to maintain consistency between the calculate metric values. Using multiple tools for different languages may result in the metrics being incomparable.

Tool	Languages Supported	Metrics Supported
Project Bauhaus [4]	Ada, C, C++, C#, Java	Lines of Code per Function, Cyclomatic McCabe Complexity, Maximal Nesting, Clone detection
Understand [5]	Ada, C/C++, C#, FORTRAN, Java, Pascal, Cobol, PHP, JavaScript, XML, Python and more	Complexity Metrics, Volume Metrics, Object Oriented Metrics
CodeSonar [6]	C, C++ and Java	Size, Complexity, Halstead metrics
SonarQube [7]	Android, C/C++, C#, Cobol, Delphi / Pascal, Erlang, Flex / ActionScript, Groovy, Java, JavaScript, PHP, PL/I, PL/SQL, Python, VB.NET, Visual Basic 6, XML and more	Complexity, Design, Documentation, Duplications, Issues, Size, Tests

Table 3: Candidate tools based on the selection criteria.

3.4.1 Outcome of the tool survey

The immediate advantage of using existing tools is that we can get the metrics data without spending a lot of time. Using SonarQube or Understand we would have a host of metrics data for many of the languages popular in open source development.

The drawbacks on the other hand, can be divided in two parts:

- **Using multiple tools from those available**
  1. Different tools use different definitions for the same metric resulting in different values.
  2. Different tools use different meta models for metric calculation, this could complicate integration.
  3. Different tools use different implementation languages, which becomes relevant when adding a new metric since users of the tools will require to learn many languages.
- **Using a single tool from those available**
  1. We assume the definition of the metrics in the tool are correct.
  2. We have no control over the model used for metrics calculation. This could pose a maintenance issue for metrics no written by the creators.

We feel the drawbacks of using existing systems outweigh the possible advantages. We thus reached the conclusion to create and implement our own meta model for metrics (discussed next) where we try to address these concerns.

[4] See `http://www.bauhaus-stuttgart.de/bauhaus/demo/index.html`
[5] See `http://www.scitools.com/`
[6] See `http://www.grammatech.com/codesonar`
[7] See `http://www.sonarqube.org/`

4 The Rascal Meta-Programming Language

4.1 The Needs of OSSMETER

As we have seen in the previous sections, to achieve the goals of OSSMETER many different metrics will have to be computed for different programming languages. We have also seen that different tools only provide partial support for the metrics that are relevant for OSSMETER. Another problem is that different tools use different versions of the same metric which makes their results hard to combine in a single framework. There is therefore a strong need for a flexible, uniform, approach to metrics calculation that can cater for all the needs of OSSMETER. Fom previous experience, we also know that directly implementing more advanced metrics in Java is error-prone and labor-intensive. This is the reason why we are introducing the state-of-the-art meta-programming language RASCAL to help solving these problems. The role of RASCAL will be:

- To formalize the Metrics Meta Model to be introduced in Section 5.
- To extract facts from source code.
- To compute metrics.
- To visualize these metrics.

4.2 Origins

RASCAL [10, 9] is a meta programming language focused on the implementation of domain-specific languages and on the rapid construction of tools for software analysis and software transformation. RASCAL is the successor to both ASF [1] and ASF+SDF [19, 18], providing features for defining grammars, parsing programs, analyzing program code, generating new programs, interacting with external tools (through Java), and visualizing the results of these operations. Given this feature set, RASCAL is an ideal language for implementing analysis tools and metrics calculations in OSSMETER.

4.3 Quick Overview

RASCAL was designed to cover the entire domain of meta-programming, shown pictorially in Figure 1. The language itself is designed with unofficial “language layers”. This allows RASCAL developers to start with just the core language features, adding more advanced features as they become more comfortable with the language. This language core contains basic data-types (booleans, integers, reals, source locations, date-time, lists, sets, tuples, maps, relations), structured control flow (if, while,

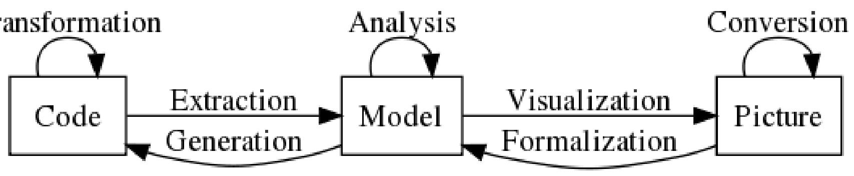


Figure 1: The meta-programming domain: three layers of software representation with transitions.

Type	**Example literal**
bool	true, false
int	1, 0, -1, 123456789
real	1.0, 1.0232e20, -25.5
rat	1r4, 22r7, -3r8
str	"abc", "first\nnext"
loc	|file:///etc/passwd|
datetime	$2012-05-08T22:09:04.120+0200
tuple[$t_1, \ldots, t_n$]	$\langle 1, 2\rangle$, $\langle$"john"$, 43, true\rangle$
list[t]	[], [1], [1,2,3], [true, 2, "abc"]
set[t]	$\{\}$, $\{1, 2, 3, 5, 7\}$, $\{$"john"$, 4.0\}$
rel[$t_1, \ldots, t_n$]	$\{\langle 1, 2\rangle, \langle 2, 3\rangle, \langle 1, 3\rangle\}$, $\{\langle 1, 10, 100\rangle, \langle 2, 20, 200\rangle\}$
map[t, u]	(), (1 : true, 2 : true), (6 : $\{1, 2, 3, 6\}$, 7 : $\{1, 7\}$)
node	f, add(x, y), g("abc", [2, 3, 4])

Table 4: Basic RASCAL Types.

switch, for), and exception handling (try, catch). The syntax of these constructs is designed to be familiar to programmers: for instance, if statements and try/catch blocks look like those found in C and Java, respectively. All data in RASCAL is immutable (i.e., no references are ever created or taken), and all code is statically typed. At this level, RASCAL looks like a standard general purpose programming language with immutable data structures.

RASCAL's type system is organized as a lattice, with bottom (`void`) and top (`value`) elements. The RASCAL `node` type is the parent of all user-defined datatypes, including the types of concrete syntax elements (`Stmt`, `Expr`, etc). Numeric types also have a parent type, `num`, but are not themselves in a subtype relation: i.e., `real` is not a parent of `int`. The basic types available in RASCAL, including examples, are shown in Table 4.

Beyond the type system and the language core, RASCAL also includes a number of more advanced features. These features can be progressively added to create more complex programs, and are needed in RASCAL to enable the full range of meta-programming capabilities. These more advanced features include:

- Algebraic data type definitions, with optional type parameters, allow the user to define new data types for use in the analysis. These data types are similar to sum types in functional languages like ML or sort and operator definitions in algebraic systems like Maude.
- A built-in grammar formalism allows the definition of context-free grammars. These grammars are used to generate a scannerless generalized parser, which allows for modular syntax definitions (i.e., unions of defined grammars) and the parsing of programs in real programming languages. The syntax formalism is EBNF-like and includes disambiguation facilities, such as the ability to indicate associativity and precedence, add follow restrictions, and even provide arbitrary code to disallow specific parses.
- Pattern matching is provided over all RASCAL data types: matches can be performed against numbers, strings, nodes, etc. A number of advanced pattern matching operators, such as deep match (`/`) (matching values nested at an arbitrary depth inside other values), negative match (`!`),

set matching, and list matching are also provided. Given the importance of concrete syntax for some meta-programming tasks, it is also possible to match against concrete syntax fragments, e.g., matching a while loop and binding variables to syntax fragments representing the loop condition and the loop body.

- Additionally, pattern matching is used in the formal parameters of functions, allowing function dispatch to be based on the pattern matching mechanism. This provides for more extensible code, since new constructors of a user-defined datatype can be handled by using new variants of an existing function, instead of requiring a single function with a large switch/case statement. As an equivalent to the switch/case `default` case, a default function provides the default behavior for the function when none of the other cases match.
- In cases where there are multiple matches for a pattern, backtracking happens from right to left in a pattern, enforcing lexical scope (names bind starting at the left, and can be used in the pattern to the right of the binding site) and providing a natural order on matches. A successful match can be explicitly discarded by the user with the `fail` keyword.
- List, set, and map comprehensions, in combination with pattern matching and other RASCAL expressions, allow new lists, sets, and maps to be constructed based on complex conditions. For instance, one could use a deep match to find all while loops in a set of program files that contain a condition with a less than comparison. Also provided is the `<-` element generation operator, which can enumerate the elements of all container data-types, e.g. lists, sets, maps, and trees, and can be used inside comprehensions and in for loops.
- String templates with margins and an auto-indent feature provide a straightforward way to generate formatted code in multi-line source code templates.
- `visit` statements, with a syntax similar to that of `switch` statements, perform *structure-shy* traversals of RASCAL data types, allowing one to match only those cases of interest. Visit cases can execute arbitrary code, for instance to keep track of statistics or analysis information, or can directly replace the matched node with one of the same type. Visits are parameterized by a traversal strategy (e.g., top-down) to allow different traversal orders.
- `solve` statements allow fixed-point computations to be expressed directly as a language construct. The statement continues to iterate as long as the result of the condition expression continues to change.

A number of RASCAL features also focus on the safety and modularity of RASCAL code. While local variable type can be inferred, parameter and return types in functions must be provided. This allows better error messages to be generated, since errors detected by the inferencer can be localized within a function, and also provides documentation (through type annotations) on function signatures. Also, the only casting mechanism is a pattern match, which prevents the problems with casts found in C (lack of safety) and Java (runtime casting exceptions). Finally, the use of persistent data structures eliminates a number of standard problems with using references which can leak out of the current scope or be captured by other variables.

```
fmod PEANO is
    sort Nat .
    op z : -> Nat [ctor] .
    op s : Nat -> Nat [ctor] .

    vars N M : Nat .

    op plus : Nat Nat -> Nat .
    eq plus(s(N),M) = s(plus(N,M)) .
    eq plus(z,M) = M .
endfm
```

Figure 2: Peano arithmetic expressed in Maude.

```
module  Nat
  data  Nat = z() | s(Nat);
  Nat plus(s(Nat n), Nat m) = s(plus(n,m));
  Nat plus(z(), Nat m) = m;
```

Figure 3: Peano arithmetic expressed in RASCAL.

4.4 Simple Examples

As a simple example, imagine that we want to work with the Peano representation of natural numbers. This can be done in any system supporting term rewriting. In Figure 2 we show has this is done in Maude [3] (in ASF it would look more or less similar). In RASCAL, this same functionality would be defined as shown in Figure 3.

Function `plus` could also be defined in RASCAL using a switch/case statement (see Figure 4).

As a more complex example, take the case where we have colored binary trees: trees with an integer in the leaves, but with a color (given as a string) defined at each composite node. This would be defined as shown in Figure 5.

Suppose we want to analyze a `ColoredTree`, computing how often each color appears at each node. The RASCAL code is shown in Listing 6. In this code, we use a map, held in a local variable `counts` with inferred type `map[str, int]`, to maintain the counts. A visit statement is used to traverse the binary tree, matching only the composite nodes, and binding the color stored in the node to the string

```
    Nat plussc(Nat n, Nat m) {
    switch (n) {
      case  s(n) :  return  s(plussc(n,m)) ;
      case  z()  :  return  m;
    }
  }
```

Figure 4: Alternative definition of `plus` using switch /case in RASCAL.

```
data  ColoredTree
        = leaf(int  n)
        | composite(str  color, ColoredTree left, ColoredTree right);
```

Figure 5: Definition of a ColoredTree datat type in RASCAL.

```
public  map [str , int ] colorDistribution(ColoredTree t) {
   counts = ();    // initialize an empty map
   visit (t) {    // all leaves and composite nodes in the tree
     case  composite(str  color,_,_):
                // for each composite node:  increment count for color
                // (use 0 as default when not yet in table)
                counts[color] ?  0  += 1;
   }
   return  counts;
}
```

Figure 6: Counting frequencies of colors in a `ColoredTree`.

variable `color`. The statement `counts[color]?0 += 1` then increments the current frequency count for the given color if it exists, or it initializes to 0 first and then increments, assigning the result back into the map entry for the color.

4.5 Some Enabling RASCAL Features in More Detail

To better understand how RASCAL can be used for computing metrics, we first discuss four key enabling RASCAL features: *type literals* that allow types to be treated as values, *source location literals* that provide access to external resources via Uniform Resource Locators (URIs), *string templates* for code generation, and the RASCAL*-to-Java bridge* to connect arbitrary Java libraries to RASCAL.

4.5.1 Type Literals

The RASCAL **type system** provides a uniform framework for both built-in and user-defined types, with the latter defined for both abstract datatypes and grammar non-terminals (also referred to as *concrete* datatypes). A built-in tree datatype (`node`) acts as an umbrella for both abstract and concrete datatypes. The type system is based on a type lattice with `void` at the bottom and `value` at the top (i.e., the supertype of all types). In between are the types for atomic values (`bool`, `int`, `real`, `str`, `loc`, `datetime`), types for tree values (`node` and defined abstract and concrete datatypes), and composite types with typed elements. Examples of the latter are `list[int]`, `set[int]`, `tuple[int,str]`, `rel[int,str]`, and, for a given non-terminal type `Exp`, `map[Exp,int]`. Sub-typing is always covariant with respect to these typed elements; with functions, as is standard, return types must be covariant, while the argument types are instead contravariant. For example, for sets, `set[int]` is a subtype of `set[value]`, while for functions, `int(value)` is a subtype of `value(int)`.

Formal type parameters allow the definition of generic types and functions. All non-atomic types can have explicit *type parameters*, written as `&T` or `&T <: Bound`. The former can be bound to any RASCAL type, the latter only to subtypes of the type `Bound`. For example, `rel[&T,&T]` defines a generic binary relation type over elements of the same type, `list[&T <: num]` defines a list with elements that can only be one of the subtypes of the type `num`, and `list[&T] reverse(list[&T] L)` defines the type of a function with the name `reverse` that returns a list with the same element type as its argument `L`.

Reified types make it possible to manipulate types as ordinary values that can be passed around, queried and manipulated. RASCAL's reification operator creates *self-describing* type values which contain both the reified type and all datatypes used in this type. A type can be reified using the prefix reification operator (#); we call such an expression a *type literal*. A reified type value contains a symbol to represent the type and a map of definitions for any abstract or concrete datatype dependencies. It is guaranteed to have the type `type[&T]`, where the type parameter `&T` is bound to the type that was reified. For example:

- `#int` produces a literal value `type(\int(),())` of type `type[int]`.
- `#rel[int,str,bool]` produces `type(\rel([\int(),\str(), \bool()]),())` of type `type[rel[int,str,bool]]`.

The `type` data constructor used to build type literals is built in to RASCAL; the representations for type symbols and their definitions are defined as RASCAL datatypes in a library module. Above, the map of definitions was empty: `()`. For abstract or concrete datatypes this map will contain the complete (possibly recursive) abstract datatype or grammar. Assume a definition for Boolean connectives:

```
  data  Bool = and(Bool l, Bool r) | t() | f();
```

then the reified type `#Bool` will produce the following term of type `type[Bool]` (some details have been elided):

```
type(adt("Bool"),
     (adt("Bool"):choice(...,constructor(adt("Bool"),"and",
           [label("l",adt("Bool")),label("r",adt("Bool"))]),...)))
```

Such self-describing type values are particularly useful in the context of defining and using meta-models.

4.5.2 Source Locations

RASCAL provides built-in support for location literals (values of type **`loc`**) that are Uniform Resource Identifiers[8] (URIs) optionally followed by text coordinates that allow the identification of specific text ranges in the information the URI points at. Location literals are quoted with bars, such as `|http://www.rascal-mpl.org|`.

[8] See `http://www.ietf.org/rfc/rfc3986.txt`.

In addition to the standard schemes like `file` (local file access) and `http` (remote file access), a number of RASCAL-special schemes are supported such as `cwd` (current working directory), `home` (the user's home directory), `std` (the RASCAL standard library), `jar` (an entry in a jar file), and `project` (an Eclipse project). The collection of schemes is openly extensible – the extension implements a contribution interface in Java.

The location datatype conveniently provides direct access to parts of the URI and gives short-hands to interact with file systems and web pages. Source locations in RASCAL are very versatile and are, for instance, used for tasks such as accessing source code locations in editors and providing hyperlinking functionality in the IDE. In the context of OSSMETER we use them to identify locations in source code.

4.5.3 String Templates and Concrete Syntax Templates

RASCAL provides both string templates and concrete syntax templates for code generation, a frequently occurring operation in meta-programming. String templates are multi-line string literals with a left-margin, interpolation of arbitrary expressions, auto-indentation, and structured control flow. For example, the following code generates the definition of a Java class named `name` with a number of fields (given as `name`×`type` pairs in relation `fields`), all indented by 2 spaces:

```
str  class(str  name, rel [str ,str ] fields) =
  "class <name> {
  '  <for  (<f,t> <- fields) {><t> <f>;
  '<}>
  '}";
```

Concrete syntax templates are parsed fragments of code, used for pattern matching and pattern construction. Concrete syntax fragments are supported for languages that have a grammar defined in RASCAL. For example:

```
import  lang::rascal::syntax::Rascal;

Module m = ‘module M imports N; ...‘;
```

The fragment within the backquotes will be parsed using the grammars defined in the current scope (here, the imported grammar of RASCAL). Concrete syntax fragments allow for anti-quoting to expand variables or to match and bind parts using pattern matching. The benefit of concrete syntax fragments is that both generated code and patterns are statically guaranteed to be syntactically correct.

4.5.4 RASCAL-to-Java Bridge

The RASCAL-to-Java Bridge makes it possible to call Java functions from RASCAL code and to build RASCAL data values in Java code. RASCAL users can extend their library reusing existing Java code or building on top of the Java standard library. This enables, for example, reuse of parsers, JDBC libraries, open Java compilers, SMT solvers, and the Apache Math library. The author of a library written in Java is responsible for producing RASCAL data of the right type. Consider the `size` function for lists:

```
@javaClass{org.rascalmpl.library.Prelude}
public java int size(list [&T] lst);
```

The modifier `java` indicates that the function `size` is written in Java and the annotation `javaClass` defines in which class the method `size` can be found. The function is then implemented by the following Java code:

```
public class Prelude {
  IValueFactory vf;
  ...
  IInteger size(IList lst) {
    return vf.integer(lst.length());
  }
}
```

The Java API `IValueFactory` makes it possible to construct arbitrary RASCAL values. If the returned type does not match the return type of the associated RASCAL function, a run-time type exception will occur, ensuring this mechanism cannot be used to break type safety. We have already used the Java Bridge to connect external PHP parsers and it will also be used in connecting RASCAL-based metrics calculators to the OSSMETER platform.

5 Metrics Meta Model (M3)

The idea behind the meta model, inspired by work on [14], [11] and [16], is to represent language specific source facts as relations in the model. Each relation in the meta model represents information that we feel is required to either calculate a metric directly or provide additional information in some metric calculation. During the formulation of the meta model, we identified relations that are exhibited by many programming languages which led us to divide the meta model into two parts, namely the *Core M3 Model* and *Language-specific M3 models* that extend the core for different programming languages. In Sections 5.2 and 5.3 we present the M3 Core Model and in Section 5.4 we discuss language-specific models and focus on Java.

An essential ingredient of our proposal are *source code locations* that are based on Uniform Resource Identifiers (URIs)[9]. An essential part of an URI is the *scheme* that defines how the information pointed to by the URI has to be interpreted. Typical examples are `http`, `ftp`, and `mailto`. In our proposal we use URI schemes to encode source language and language-element that the URI points to. Examples are:[10]

[9] See `http://tools.ietf.org/html/rfc3986`.

[10] We use the syntax for source locations as provided by the `loc` datatype in the RASCAL language, see Section 4.5.2.

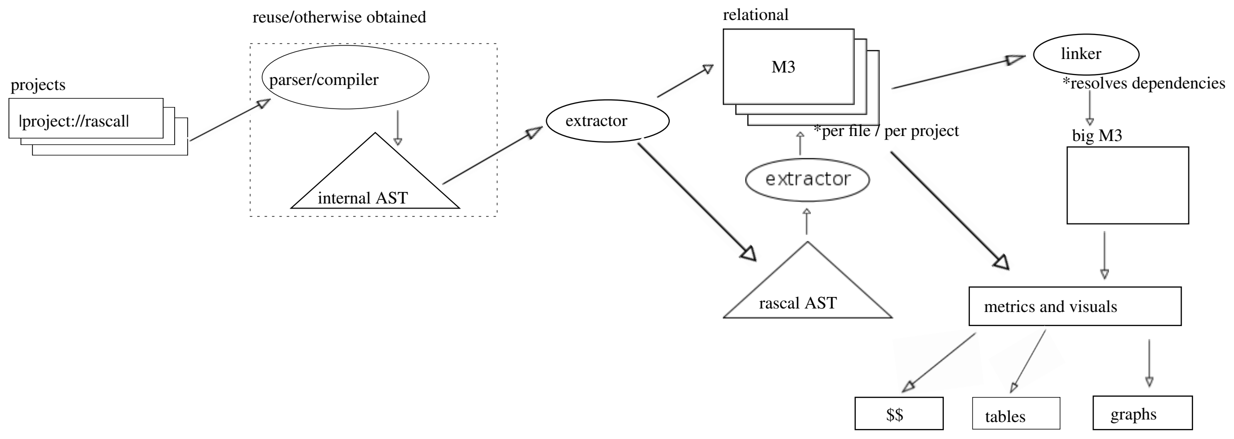


Figure 7: Reference Architecture

- `|java+class://P2SnakesLadders/snakes/Game|`: defines Java as source language and denotes a *class* declaration.
- `|java+method://P2SnakesLadders/snakes/Game/setSquare(int,snakes.ISquare)|`: defines Java as source language and denotes a *method* declaration.
- `|java+field://P2SnakesLadders/snakes/Game/squares|`: defines Java as source language and denotes a *field* declaration.
- `|project:// P2SnakesLadders/src/snakes/Game.java|(3200,53,<130,41>,<130,94>)` shows how specific source coordinates (line and character information) can be included in source locations.

In addition to providing a completely general and extensible naming scheme, it is also straightforward to provide IDE support for the hyperlinking between extracted data and the original source code.

5.1 Architecture

Figure 7, shows how the M3 meta-model is created from source files and provides an indication of how the M3 model will be used by metrics/visualizations. The first task is to extract source facts in the form of an M3 model. We achieve this through reusing parser or compiler for each programming language encountered in the project. For each language supported in OSSMETER, we will need custom extractors. The extractor traverses through the internal Abstract Syntax Tree (AST) representation of each source file in the project, creating the relations in the model and later fuses them together as a single model for a project. The granularity of the M3 model can be per file or per project. In the case the granularity is set to be a project, we get a single M3 model even if multiple languages are found to be present with the difference between the languages being represented in the scheme of the URI's we use to represent source code elements.

The details of how the model can be used to implement metrics will be discussed in Section 5.5.1.

5.2 The Core M3 Model

The definition of a model contains an "id" which defines the project/file for which the model is being created. We then add the following relations to the core model.

- **Declarations**. The declarations relation maps declared language elements to their physical location in a file.
- **Uses**. The uses relation maps uses of declared elements to their physical location in a file.
- **Containment**. The containment relation maps the elements that logically contain other elements to define a structure. For example, in Java files contain classes, classes contain fields and methods etc.
- **Names**. The names relation maps a declared name of an element to the qualified name that the compiler uses to uniquely identify it.
- **Documentation**. The documentation relation contains comments that are mapped to the to a physical location in a file.

The model also contains a list to store compiler generated error/warning messages.

5.3 The Core M3 Model in Rascal

The core relations of the M3 model are represented by a central (empty) model (`M3`) to which the relevant relations are attached (using Rascal's annotation operator `@`):

- `M3@declarations`: maps declarations to where they are declared. contains any kind of data or type or code declaration (classes, fields, methods, variables, etc. etc.).
- `M3@types`: assigns types to declared source code artifacts.
- `M3@uses`: maps source locations of usages to the respective declarations.
- `M3@containment`: what is logically contained in what else (not necessarily physically, but usually also).
- `M3@messages`: error messages and warnings produced while constructing a single M3 model.
- `M3@names`: convenience mapping from logical names to end-user readable (GUI) names, and vice versa.
- `M3@documentation`: comments and javadoc attached to declared things
- `M3@modifiers`: modifiers associated with declared things.

The definition of the M3 Core Model in Rascal:

```
data  M3 = m3(loc  id);
anno  rel [loc  name, loc  src]                 M3@declarations;
anno  rel [loc  src, loc  name]                 M3@uses;
anno  rel [loc  from, loc  to]                  M3@containment;
anno  list [Message messages]                   M3@messages;
anno  rel [str  simpleName, loc  qualifiedName]  M3@names;
anno  rel [loc  definition, loc  comments]      M3@documentation;
```

The core model will only contain facts that are language independent. In addition to these basic facts that we expect to encounter in any type of programming language, we can easily add new relations to the model to incorporate any language independent facts we may find.

5.4 Language Specific M3 Model

Each language we provide support for in the OSSMETER platform will have it own M3 model. The language specific model will add new relations to the Core M3 model that will be relevant for metric calculation. As an example we provide Java M3 Model.

5.4.1 M3 Java Model

The M3 Java Model extends the M3 Core Model and adds the following Java-specific relations:

- **Extends**. Contains the extends relation in Java between classes/interfaces.
- **Implements**. Contains the implements relation between classes and interfaces.
- **MethodInvocations**. Contains all the methods that are called from a method.
- **FieldAccess**. Contains any access to a Java field from anywhere in the source code.
- **TypeDependency**. Contains all the types that a Java element depends on.
- **MethodOverrides**. Contains the relations between methods and the methods that they override.

5.4.2 The M3 Java Model in Rascal

The M3 Core Model is extended with the following Java-specific relations:

- `M3@extends`: classes extending classes and interfaces extending interfaces.
- `M3@implements`: classes implementing interfaces.
- `M3@methodInvocation`: methods calling each other (including constructors).
- `M3@fieldAccess`: code using data (like fields).
- `M3@typeDependency`: using a type literal in some code (types of variables, annotations).
- `M3@methodOverrides`: which method override which other methods

The definition of the M3 Java Model in Rascal:

```
extend  m3::Core;

anno  rel [loc  from, loc  to] M3@extends;

anno  rel [loc  from, loc  to] M3@implements;

anno  rel [loc  from, loc  to] M3@methodInvocation;

anno  rel [loc  from, loc  to] M3@fieldAccess;
anno rel [loc from, loc to] M3@typeDependency;
anno rel [loc from, loc to] M3@methodOverrides;
```

An extract of the M3 model for a sample Java project can be found in Appendix A.

5.5 Metrics based on M3 model

The advantage of the metrics based on the M3 model are two fold:

- Metric calculation is abstracted to a higher level and becomes language independent.
- We can change the granularity of the source code facts with ease. The M3 model can be created for a file, folder, project or any combination of these as required and the metric calculation will not need to change.

For use in the OSSMETER project, the platform (WP5) will host all the implemented metrics. The platform contains a Rascal interpreter that will first calculate the M3 model for each project and then pass the model to all the metrics available in its store. Each metric will be identified to the platform using URI's of the form "`metrics://module/metric`", where "`module`" will be the Rascal module where the metric definition is present and "`metric`" the name of the Rascal function of the metric. The scheme has been chosen as "`metrics`" for now but may change in the future to represent metric types like "`oometrics`" for example. This approach allows the users with an easy means to implement their own metric. The user will only need to accept the M3 model (if needed), perform the tasks in the function and return the result back to the interpreter which will store it accordingly. The users only need to focus on what data the metric needs from the model and how it is calculated.

5.5.1 Metric Implementation

Metrics will be implemented as functions in Rascal. Each metric will need to accept the M3 model for which the metric is to be calculated as an argument. The process of implementing a metric consists of extracting relevant information from the model, analyzing the information and reporting the calculated values.

5.5.2 Examples of some metric implementations

In this section, we present some metrics that we have calculated using the M3 model. These metrics are already useful in their own right, but should primarily be seen as illustration how well the M3 model can provide the basic data for metrics calculations.

5.5.2.1 Number of Methods

The number of methods (NOM) metric is simply a count of the methods that are declared in classes. In order to calculate this metric using the M3 model, we first extract the information relevant to this metric i.e., relations which specify which classes define which methods using the convenience function `declaredMethods`, convert the relations into a map with the class names as the key and return a map which contains the count with the class names as key and the NOM as its value.

This metric counts all metrics related to the NOM metric like number of public method, number of private methods. We only require to pass the set of modifiers that we are interested in as a parameter to get only the count for those methods. If the modifiers parameter is an empty set, we get the NOM metric.

```
public  map [loc , int ] NOM(M3 model, set [Modifier] modifiers = {}) {
  classMethods = declaredMethods(model, checkModifiers = modifiers);
  classMethodsMap = toMap(classMethods);
  return  (class :  size(classMethodsMap[class]) | class <- classMethodsMap);
}
```

5.5.2.2 Lines of code

The following lines of code (LOC) metrics are supported by the model. These are all convenience functions that iterate through all the physical files used to create the model, count their respective LOC and sum them up to return the result. For each of the LOC metric, we use the convenience function `files` which extracts physical file locations from a model.

- **Physical lines of code**: Counts all the lines in the source files.

  ```
  int  countProjectTotalLoc(M3 model) =
       (0  | it  + countFileTotalLoc(model, cu) | cu <- files(model));
  ```

- **Commented lines of code**: Counts only the lines in comments.

  ```
  int  countProjectCommentedLoc(M3 model) =
       (0  | it  + countCommentedLoc(model, cu) | cu <- files(model));
  ```

- **Empty lines of code**: Count only empty lines.

  ```
  int  countProjectEmptyLoc(M3 model) =
       (0  | it  + countEmptyLoc(model, cu) | cu <- files(model));
  ```

- **Source lines of code**: Counts all lines except the commented and empty lines

  ```
  int  countProjectSourceLoc(M3 model) =
       countProjectTotalLoc(model) -  countProjectCommentedLoc(model) -
       countProjectEmptyLoc(model);
  ```

- **Physical lines of code per language**: Counts all lines in the source files and classisfies them according to their proramming language. The languages are determined from the scheme of each file.

```
map [str  language, int  count] countTotalLocPerLanguage(M3 model) {
  map [str , int ] result = ();
  for  (cu <- files(model)) {
    str  lang = split("+" , cu.scheme)[0 ];
    result[lang] ?  0  += countFileTotalLoc(model, cu);
  }
  return  result;
}
```

- **Source lines of code per language**: Counts source lines and classifies them according to their programming language.

```
map [str  language, int  count] countSourceLocPerLanguage(M3 model) {
  map [str , int ] result = ();
  for  (cu <- files(model)) {
    str  lang = split("+" , cu.scheme)[0 ];
    result[lang] ?  0  += countSourceLoc(model, cu);
  }
  return  result;
}
```

5.5.3 Assessment

The goal of creating the M3 model was to make metric calculation easy and generic. For each metric we want to calculate, we pass it the full M3 model. It is up to the metric to decide what information it requires from the model to calculate its value.

The number of methods (NOM) metric example provided above can be used as a case to understand how metric calculation becomes generic. As we can see, the NOM metric first queries the model and extracts relations which map methods to the classes they are defined in. In object oriented languages where this type of relation exists, we would receive the mapping. In case the model was created for a non-object oriented language, we would receive an empty set here and the metric would return an empty map signifying that the NOM metric doesn't apply to the model.

The model is also easily extensible, with changes being localized. If we need to add support for a language, we extend the core for that language and any metric that applies to the language will be calculated. If we need to add support for a new metric, the metric needs only ask the information it requires from the model and it becomes available to all languages.

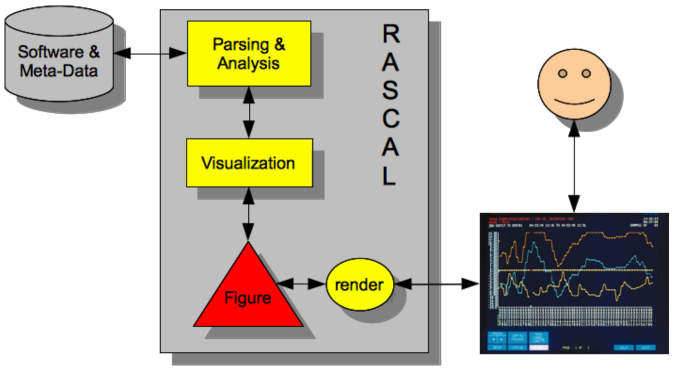


Figure 8: Overview of the RASCAL visualization library.

6 Visualization

The mapping of source code facts to relations in the M3 model can be easily visualized using RASCAL's visualization libraries. In this section, we present – as a proof-of-concept – two visualizations created using the M3 model and the RASCAL visualization library. We expect that in later deliverables we will also explore web-based visualizations.

6.1 The Rascal Visualization Library

The RASCAL visualization library [8] has as aim to provide a software visualization framework that:

- enables non-experts to easily create, combine, extend and reuse interactive software visualizations;
- integrates seamlessly with existing techniques for software analysis (parsing, pattern matching, tree traversal, constraint solving) and software transformation (rewriting, string templates).

Our main objective is to liberate the creator of visualizations from many low-level chores, such as programming of explicit coordinates, mapping metrics related to software properties to sizes of shapes and figures, and to provide high-level features instead, like figure composition, fully automatic figure placement and symbolic links between visualizations elements. Since RASCAL already provides excellent facilities for software analysis and transformation, the main challenge is therefore to provide a software visualization framework that integrates well with and provides full access to what RASCAL has to offer. The contributions of this library can be summarized as follows:

- A compositional, coordinate-free, visualization framework that provides primitives for drawing elementary shapes and composite figures.
- Mechanisms to associate numeric scales with arbitrary figures.
- The first attempt we are aware of to decompose charts into reusable primitives.
- The integration of this framework with the RASCAL language and infrastructure thus creating a true "One-Stop-Shop" for software analysis, transformation and visualization.
- An analysis of the software visualization domain that can form the basis for a domain-specific language for software visualization.

The technical architecture of our visualization framework is shown in Figure 8. The given *Software & Meta-Data* is first parsed and then relevant analyses are performed (*Parsing & Analysis*). Parsing and analysis can be completely defined and arbitrary languages and data formats can therefore be parsed and analyzed. The analysis results are then turned into a figure (*Visualization*) and the result is an instance of the `Figure` data type, an ordinary RASCAL datatype that is used to represent our visualizations. Note, for later reference, that another data type, `FProperty`, is used to represent all possible visual properties of Figures. Figures are interpreted by a render function that transforms them in an actual on-screen display with which the user can interact. There is *two-way communication* between visualization and user: the visualization functions create a figure that is shown to the user, but this figure may contain call backs (RASCAL functions) that can be activated by user actions like pointing, hovering, selecting or scrolling.

6.2 Examples of Metrics Visualizations

Creating a visualization always consists of the following steps:

- Transform the given source code or metrics data into a value of type `Figure`.
- Display this `Figure` value on the screen using the `render` function.

A `Figure` can be composed of:

- Primitive figures like `text`, `outline`, `box` and `ellipse`.
- Basic composition operators like `hcat` (horizontal composition), `vcat` (vertical composition), `overlay` (superposition), and `grid` (placement in a rectangular grid).
- Advanced composition operators like `pack` (bin packing in minimal space), `graph` (hierarchical or spring-like graph), `tree` (tree structure), and `treemap` (treemap).
- Properties like `lineWidth`, `lineColor`, `fillColor`, and alignment (`halign`, `valign`), spacing (`gap`, `hgap`, `vgap`) and relative size (`grow`, `shrink`). Properties are also used to describe interactive aspects of a visualization. For instance `onMouseEnter` defines a call-back function to be called when the mouse enters this specific figure element.

6.2.1 Visualizing a Type Hierarchy

In Figure 9, we visualize the type hierarchy of a sample Java program (the same programs whose M3 Model is given in Appendix A) as a tree. The light blue circles represent classes and dark blue circles represent interfaces. The top of the hierarchy represent the Java Object class. The RASCAL code to create this visualization is shown in Figure 10.

6.2.2 Visualizing the Structure of a Java Program

In Figure 11, we visualize the complete structure of the same Java program as a graph. The various language elements are color-coded as described in the legend. The RASCAL code to create this visualization is shown in Figure 12.

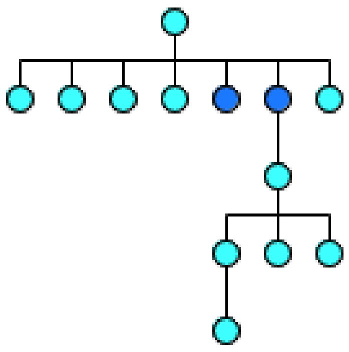


Figure 9: Type hierarchy for a sample Java program visualized as tree.

```
public  void  visualizeAsTree(rel [loc , loc ] classRels, loc  rootNode) {
    list [Figure] called =  [];
    list [Figure] callers = [];
    treeRoot = ellipse(size(10), fillColor(color(getColor(rootNode.scheme))),
                             onMouseEnter(void () {output = toString(rootNode);}));
    for (root <- successors(classRels, rootNode)) {
        called += addTreeNode(root, classRels, true );
    }
    for (root <- predecessors(classRels, rootNode)) {
        callers += addTreeNode(root, classRels, false );
    }
    t2 = vcat([box(text(str () {return  output;}), shrink(0.25, 0.1), left(), top()),
              tree(treeRoot,
                   [ tree(ellipse(size(10), fillColor(color("black" , 0.5)),
                          onMouseEnter(void () { output = "children" ; })), called),
                     tree(ellipse(size(10), fillColor(color("black" , 0.5)),
                          onMouseEnter(void () { output = "parents" ; })), callers)
                   ], left())], vgap(20), std(left()), std(hgap(10)), std(vgap(20)));
    render(t2);
}
```

Figure 10: Source code for Figure 9.

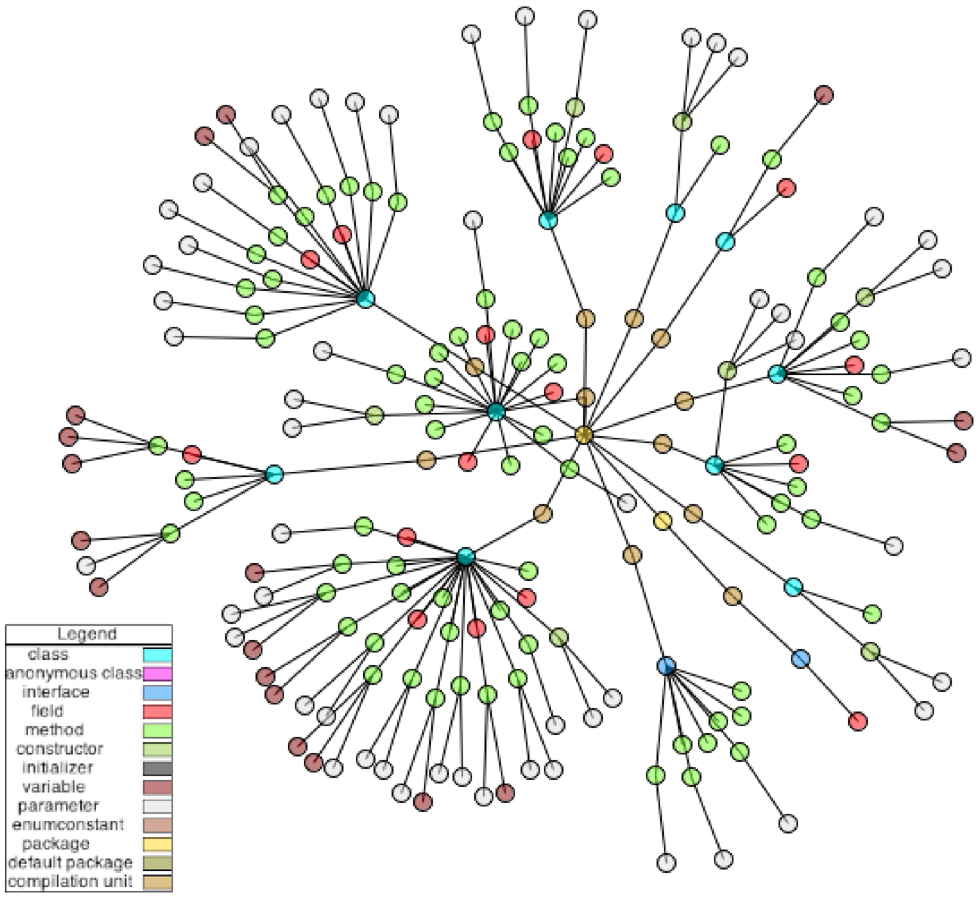


Figure 11: A complete Java program visualized as a graph.

```
public  void  visualizeAsSpring(rel [loc , loc ] classRels, loc  root) {
    classRels = getSubGraph(classRels, root);
    Figures nodes = [];
    list [Edge] edges = [];
    for (nodeLoc <- carrier(classRels))
        if (nodeLoc == root) {
           nodes += [ ellipse(id(getID(nodeLoc)), size(10, 10),
                                fillColor(color(getColor(nodeLoc.scheme), 0.5))) ];
        } else  {
           nodes += [ ellipse(id(getID(nodeLoc)), size(10, 10),
                                fillColor(color(getColor(nodeLoc.scheme), 0.5))) ];
        }
    for (<relLHS, relRHS> <- classRels) {
        edges += [ edge(getID(relLHS), getID(relRHS)) ];
    }
    render(hcat([graph(nodes, edges, hint("spring" ), size(1000))]));
}
```

Figure 12: Source code for Figure 11.

Expected Compliance	Description
Full Compliance	Full compliance is expected.
Partial Compliance/Not yet known	Partial compliance is expected or expected compliance is not yet known.
No Compliance	No compliance is expected.

Table 5: Coding scheme for expected requirement compliance.

7 Expected Compliance with OSSMETER Requirements

Given the domain analysis in the previous sections, we can now summarize the requirements on WP3 and give our assessment how well these requirements can eventually be met. We use the coding scheme shown in Table 5.

The requirements for WP 3 are as follows:

ID	Requirement	Priority	Expected compliance
13	Metrics for software quality shall be defined that are independent of any programming language (language-agnostic metrics).	SHALL	Full Compliance
14	Fact extractors shall be available that extract from source code the facts that are needed for computing language-agnostic metrics.	SHALL	Full Compliance
15	Language-specific metrics for software quality shall be defined for Java.	SHALL	Full Compliance
16	The facts needed to compute language-specific metrics for Java shall be extracted.	SHALL	Full Compliance
17	Language-specific metrics for software quality may be defined for other languages (PHP, Python, C).	MAY	Partial Compliance/Not yet known. We plan to define PHP metrics.
18	The facts needed to compute language-specific metrics for other languages (PHP, Python, C) may be extracted.	MAY	Partial Compliance/Not yet known. We plan to extract PHP metrics.
19	Calculation of software quality metrics should, where possible, be the same across all languages and paradigms.	SHOULD	Full Compliance
20	Development activity shall be measured by the number of committed changes.	SHALL	Full Compliance
21	Development activity shall be measured by the size of committed changes.	SHALL	Full Compliance

22	Development activity may be measured by the distribution of active committers.	MAY	Full Compliance
23	Development activity may be measured by the ratio between old and new committers.	MAY	Full Compliance
24	History of some metrics should be captured to summarize quality evolution during development.	SHOULD	Full Compliance
25	A model shall be designed to represent quality and activity metrics.	SHALL	Full Compliance
34	Provide a rating of the quality of code comments of the OSS project	SHALL	Partial Compliance/Not yet known. We have not yet investigated how to define such a metric.
35	Provide a well-structured code index for the OSS project	SHALL	Full Compliance
36	Provide a rating of the use of advanced language features for the OSS project	SHOULD	Full Compliance. The list of such features still has to be designed.
37	Provide a rating of the use of testing cases for the OSS project	SHALL	Partial Compliance/Not yet known. In principle, test cases have to be executed to determine this. We have to explore how a weaker, but meaningful, metric can be defined.
38	Provide an indicator of the possible bugs from empty try/catch/finally/switch blocks for the OSS project	SHALL	Full Compliance
39	Provide an indicator of the dead code from unused local variables, parameters and private methods for the OSS project	SHALL	Partial Compliance/Not yet known. Exhaustive dead code detection requires costly data flow analysis; we still have to decide whether this is feasible or that an approximation will be used instead.
40	Provide an indicator of the empty if/while statements for the OSS project	SHALL	Full Compliance
41	Provide an indicator of overcomplicated expressions from unnecessary if statements and for loops that could be while loops for the OSS project	SHALL	Full Compliance
42	Provide an indicator of suboptimal code from wasteful String/StringBuffer usage for the OSS project	SHALL	Partial Compliance/Not yet known. The precise definition of ’wastefull" is still to be determined. This determines the achievable compliance.
43	Provide an indicator of duplicate code by detecting copied/pasted code for the OSS project	SHALL	Full Compliance

44	Provide an indicator of the use of Javadoc comments for classes, attributes and methods for the OSS project	SHALL	Full Compliance
45	Provide an indicator of the use of the naming conventions of attributes and methods for the OSS project	SHALL	Full Compliance
46	Provide an indicator of the limit of the number of function parameters and line lengths for the OSS project	SHALL	Full Compliance
47	Provide an indicator of the presence of mandatory headers for the OSS project	SHALL	Partial Compliance/Not yet known. The implications of this requirements have to be further explored.
48	Provide an indicator of the use of packets imports, of classes, of scope modifiers and of instructions blocks for the OSS project	SHALL	Full Compliance
49	Provide an indicator of the spaces between some characters for the OSS project	SHALL	Partial Compliance/Not yet known. The implications of this requirements have to be further explored.
50	Provide an indicator of the use of good practices of class construction for the OSS project	SHALL	Partial Compliance/Not yet known. The implications of this requirements have to be further explored.
51	Provide an indicator of the use of multiple complexity measurements, among which expressions for the OSS project	SHALL	Full Compliance.
52	Provide an indicator of the cyclomatic complexity for the OSS project	SHALL	Full Compliance

8 Summary, Conclusions and Future Work

8.1 Summary

In this deliverable we have obtained the following results:

- In Section 2 we have explored how to design a quality model for OSSMETER.
- In Section 3 we have described essential quality attributes for OSS projects.
- In Section 3 we have also described our initial selection of metrics for measuring these quality attributes. This section also presents a brief tool survey.
- In Section 4 we have given a quick overview of the RASCAL meta-programming language that will be used for fact extraction and metric calculation.
- In Section 5 we have described M3 (Metrics Meta-Model) that we have designed for representing facts and metrics in OSSMETER. The core M3 model and an extension for Java have been presented.
- In Section 6 we have briefly illustrated the possibilities for visualizing facts and metrics that are stored in an M3 model.
- In Section 7 we have summarized the compliance of the solutions proposed in this deliverable with the overall OSSMETER requirements.

8.2 Answers to Questions in Workplan

In the workplan for Task 3.1 several questions were raised. Here we summarize how these questions have been answered in this deliverable.

8.2.1 Which source code and version management attributes are most relevant for tracking the quality of OSS projects?

This question has been answered in Section 3.

8.2.2 What are the observable effects of these attributes in the source code and the version management system?

This question has been answered in Section 3.

8.2.3 How can these attributes be measured?

This question has been answered in Section 3.

8.2.4 What are the costs of measurement tools for each metric (= cost of measurement implementation)?

There are severals costs involved here:

- The costs of including a new tool in the platform.
- The costs of solving incompatibilities in metric calculation between different tools.
- The costs of actually implementing a new metric.

As we have argued in Section 4, these costs are high and we propose to reduce them by relying on a common implementation of all metrics in the meta-programming language RASCAL. This leads to as much uniformity and reuse as possible. Initial experience shows that, once the M3 model is in place, metrics calculation becomes much cheaper using this approach. We only give qualitative arguments to support this choice.

8.2.5 What are the tradeoffs between accuracy and efficiency for each metric? (= cost of measurement application?)

We have not specifically addressed this question since it can only be answered given specific metrics. We will report on this in the next deliverable.

8.2.6 What tools (existing or new) are needed to provide the required measurements?

As we have argued in Section 4, the existing tools and metrics combination do not adequately form a cost effective solution for OSSMETER, thus we propose to use RASCAL to provide the required measurements.

8.3 Conclusions and Future Work

We can conclude that the objectives for this deliverable as sketched in Sections 1 and 1.1 have been met by the design and experiments described here. We believe that we have created a flexible and versatile foundation for representing extracted facts and computing metrics.

However, this foundation has to be turned into a practically usable infrastructure by the following future steps:

- Integration with the MongoDB-based infrastructure provided by WP5. The integration of the proof of concept metrics with the WP5 infrastructure is under way. This should provide us with an idea of the information that WP5 must provide WP3 with.
- Assessment of performance and identifying opportunities for optimization.
- Experimentation with the already implemented metrics.
- Implementation of new metrics.
- Creation of new M3 models for new languages like, for instance, PHP.
- Experimentation on the presentation of the metrics data to the users.

- Exploration of the possible visualizations (of source facts and in some cases metrics).
- Experimentation on the possibility of creating threshold values for the quality of open source projects.

References

[1] J.A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification*. ACM Press/Addison-Wesley, 1989.

[2] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[3] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *Maude Manual (Version 2.4)*. SRI International, Menlo Park, CA, October 2008. Revised February 2009.

[4] F. Brito e Abreu. The mood metrics set. *ECOOP 95 Workshop on Metrics*, 1995.

[5] N. Fenton and M. Neil. Software metrics: Successes, failures, and new directions. *Journal of Systems and Software*, 47:149–157, 1999.

[6] N.E. Fenton. *Software Metrics, A Rigorous Approach*. Chapman & Hall, 1991.

[7] I. Heitlager, T. Kuipers, and J. Visser. A practical model for measuring maintainability. In *International Conference on the Quality of Information and Communications Technology (QUATIC'07)*, pages 30–39, 2007.

[8] Paul Klint, Bert Lisser, and Atze van der Ploeg. Towards a one-stop-shop for analysis, transformation and visualization of software. In Anthony M. Sloane and Uwe Aßmann, editors, *SLE*, volume 6940 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2011.

[9] Paul Klint, Tijs van der Storm, and Jurgen Vinju. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Proceedings of SCAM'09*, pages 168–177. IEEE, 2009.

[10] Paul Klint, Tijs van der Storm, and Jurgen Vinju. EASY Meta-programming with Rascal. In *Post-proceedings of GTTSE'09*, volume 6491 of *LNCS*, pages 222–289. Springer, 2011.

[11] Adrian Kuhn and Toon Verwaest. FAME, a polyglot library for metamodeling at runtime. *In Workshop on Models at Runtime*, pages 57–66, 2008.

[12] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics*. Prentice Hall, 1994.

[13] T.J. McCabe. A complexity measure. *IEEE Trans. Software Eng.*, SE-2:308–320, 1976.

[14] S. Demeyer S. Tichelaar, S. Ducasse and O. Nierstrasz. A meta-model for language-independent refactoring. In *Proc. Int'l Sym. Principles of Software Evolution*, pages 157–169. IEEE Computer Society, 2000.

[15] I. Samoladas, G. Gousios, D. Spinellis, and I. Stamelos. The sqo-oss quality model: measurement based open source software evaluation. In *Proceedings of the International Conference on Open Source Systems 2008, Open Source Development, Communities and Quality*, pages 237–248, 2008.

[16] M. Scotto, A. Sillitti, G. Succi, and T. Vernazza. A relational approach to software metrics. In *Proceedings of the Software Applied Computing (SAC'2004)*, pages 1536–1540, March 2004.

[17] Jeremy Singer, Gavin Brown, Mikel Lujn, Adam Pocock, and Paraskevas Yiapanis. Fundamental nano- patterns to characterize and classify java methods. In *Electronic Notes in Theoretical Computer Science, Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009)*, pages 191–204, 2010.

[18] M.G.J. van den Brand, M. Bruntink, G.R. Economopoulos, H.A. de Jong, P. Klint, T. Kooiker, T. van der Storm, and J.J. Vinju. Using The Meta-environment for Maintenance and Renovation. In *Proceedings of CSMR'07*, pages 331–332. IEEE, 2007.

[19] M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *Proceedings of CC '01*, volume 2027 of *LNCS*, pages 365–370. Springer, 2001.

[20] Eva van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *In Reverse Engineering, 2002. Proceedings. Ninth Working Conference*, pages 97–106, 2002.

Appendix A An extract of the M3 model for a Java program

Here we present a concrete instance of an M3 Java Model and show an extract of the M3 representation of a small Java project. Later, in Section 6, we will use this same project to generate the visualizations in Figures 9 and 11. The listing only provides partial information of all the relations.

The first line defines the model that we are creating. In this case, it states that we are creating the M3 model for the project `P2SnakesLadders`. Each relation available to the model starts with "`@`" in the text like `@fieldAccess` or `@extends`.

If we look at a single tuple in the relation `extends`, say

```
< |java+class://P2SnakesLadders/snakes/Snake|,
|java+class://P2SnakesLadders/snakes/Ladder| >
```

we interpret it as follows: there is a class `Snake` in package `snakes` in project `P2SnakesLadders` that extends `Ladder` in the same package of the same project.

```
m3(|project://P2SnakesLadders|)[
  @fieldAccess={
    <|java+method://P2SnakesLadders/snakes/Player/moveForward(int)|,|java+field://
        P2SnakesLadders/snakes/Player/square|>,
    <|java+method://P2SnakesLadders/snakes/Square/previousSquare()|,|java+field://
        P2SnakesLadders/snakes/Square/position|>,
    <|java+method://P2SnakesLadders/snakes/SimpleGameTest/initialStrings(snakes.
        Game)|,|java+field://P2SnakesLadders/snakes/SimpleGameTest/jack|>,
    <|java+method://P2SnakesLadders/snakes/Game/currentPlayer()|,|java+field://
        P2SnakesLadders/snakes/Game/players|>,
    <|java+variable://P2SnakesLadders/snakes/Die/roll()/result|,|java+field://
        P2SnakesLadders/snakes/Die/FACES|>
    ...},
  @extends={
    <|java+class://P2SnakesLadders/snakes/Snake|,|java+class://P2SnakesLadders/
        snakes/Ladder|>,
    <|java+class://P2SnakesLadders/snakes/FirstSquare|,|java+class://
        P2SnakesLadders/snakes/Square|>,
    <|java+class://P2SnakesLadders/snakes/LastSquare|,|java+class://P2SnakesLadders
        /snakes/Square|>,
    <|java+class://P2SnakesLadders/snakes/Ladder|,|java+class://P2SnakesLadders/
        snakes/Square|>
  },
  @methodInvocation={
    <|java+method://P2SnakesLadders/snakes/SimpleGameTest/move1jack(snakes.Game)|,|
        java+method://P2SnakesLadders/org/junit/Assert/assertEquals(java.lang.Object
        ,java.lang.Object)|>,
    <|java+method://P2SnakesLadders/snakes/Game/play(snakes.Die)|,|java+method://
        P2SnakesLadders/snakes/Game/winner()|>,
    <|java+method://P2SnakesLadders/snakes/Game/setSquare(int,snakes.ISquare)|,|
        java+method://P2SnakesLadders/snakes/Game/getSquare(int)|>,
    <|java+method://P2SnakesLadders/snakes/Player/moveForward(int)|,|java+method://
        P2SnakesLadders/snakes/ISquare/leave(snakes.Player)|>,
    <|java+method://P2SnakesLadders/snakes/Player/wins()|,|java+method://
        P2SnakesLadders/snakes/ISquare/isLastSquare()|>
    ...},
  @typeDependency={
    <|java+method://P2SnakesLadders/snakes/SimpleGameTest/move6jill(snakes.Game)|,|
        java+interface://P2SnakesLadders/ch/unibe/jexample/Given|>,
    <|java+variable://P2SnakesLadders/snakes/DieTest/reached(int)/i|,|java+
        primitiveType://P2SnakesLadders/int|>,
    <|java+method://P2SnakesLadders/snakes/DieTest/testInRange()|,|java+interface
        ://P2SnakesLadders/org/junit/Test|>,
    <|java+field://P2SnakesLadders/snakes/Game/winner|,|java+class://
        P2SnakesLadders/snakes/Player|>,
    <|java+variable://P2SnakesLadders/snakes/SimpleGameTest/newGame()/args|,|java+
        array://P2SnakesLadders/snakes/Player%5B%5D|>
    ...},
  @messages=[],
  @containment={
    <|java+method://P2SnakesLadders/snakes/SimpleGameTest/newGame()|,|java+variable
        ://P2SnakesLadders/snakes/SimpleGameTest/newGame()/game|>,
    <|java+class://P2SnakesLadders/snakes/Snake|,|java+method://P2SnakesLadders/
        snakes/Snake/squareLabel()|>,
    <|java+class://P2SnakesLadders/snakes/Player|,|java+method://P2SnakesLadders/
        snakes/Player/toString()|>,
    <|java+package://P2SnakesLadders/snakes|,|java+compilationUnit://
        P2SnakesLadders/src/snakes/ISquare.java|>,
    <|java+class://P2SnakesLadders/snakes/FirstSquare|,|java+field://
        P2SnakesLadders/snakes/FirstSquare/players|>
  ...},
  @names={
    <"move8jillWins",|java+method://P2SnakesLadders/snakes/SimpleGameTest/
        move8jillWins(snakes.Game)|>,
    <"java",|java+package://P2SnakesLadders/java|>,
    <"square",|java+variable://P2SnakesLadders/snakes/Game/addSquares(int)/square
        |>,
    <"position",|java+parameter://P2SnakesLadders/snakes/Game/setSquareToSnake(int,
        int)/position|>,
    <"out",|java+field://P2SnakesLadders/java/lang/System/out|>
  ...},
  @implements={<|java+class://P2SnakesLadders/snakes/Square|,|java+interface://
      P2SnakesLadders/snakes/ISquare|>},
@documentation={
  <|java+compilationUnit://P2SnakesLadders/src/snakes/Game.java|,|project://
     P2SnakesLadders/src/snakes/Game.java|(3200,53,<130,41>,<130,94>)>,
  <|java+compilationUnit://P2SnakesLadders/src/snakes/Game.java|,|project://
     P2SnakesLadders/src/snakes/Game.java|(1426,23,<62,30>,<62,53>)>,
  <|java+compilationUnit://P2SnakesLadders/src/snakes/Game.java|,|project://
     P2SnakesLadders/src/snakes/Game.java|(1594,54,<70,2>,<70,56>)>,
  <|java+compilationUnit://P2SnakesLadders/src/snakes/Game.java|,|project://
     P2SnakesLadders/src/snakes/Game.java|(3310,42,<132,23>,<132,65>)>,
  <|java+compilationUnit://P2SnakesLadders/src/snakes/Game.java|,|project://
     P2SnakesLadders/src/snakes/Game.java|(709,4,<33,0>,<33,4>)>
...},
@uses={
  <|project://P2SnakesLadders/src/snakes/DieTest.java|(444,7,<26,13>,<26,20>),|
     java+method://P2SnakesLadders/snakes/DieTest/reached(int)|>,
  <|project://P2SnakesLadders/src/snakes/Game.java|(1470,4,<63,20>,<63,24>),|java
     +method://P2SnakesLadders/snakes/Player/wins()|>,
  <|project://P2SnakesLadders/src/snakes/DieTest.java|(307,5,<15,43>,<15,48>),|
     java+field://P2SnakesLadders/snakes/Die/FACES|>,
  <|project://P2SnakesLadders/src/snakes/SimpleGameTest.java
     |(2668,4,<96,18>,<96,22>),|java+field://P2SnakesLadders/snakes/
     SimpleGameTest/jack|>,
  <|project://P2SnakesLadders/src/snakes/Square.java|(728,10,<38,14>,<38,24>),|
     java+method://P2SnakesLadders/snakes/Square/isOccupied()|>
...},
@methodOverrides={
  <|java+method://P2SnakesLadders/snakes/FirstSquare/isOccupied()|,|java+method
     ://P2SnakesLadders/snakes/Square/isOccupied()|>,
  <|java+method://P2SnakesLadders/snakes/Player/toString()|,|java+method://
     P2SnakesLadders/java/lang/Object/toString()|>,
  <|java+method://P2SnakesLadders/snakes/FirstSquare/leave(snakes.Player)|,|java+
     method://P2SnakesLadders/snakes/ISquare/leave(snakes.Player)|>,
  <|java+method://P2SnakesLadders/snakes/FirstSquare/isFirstSquare()|,|java+
     method://P2SnakesLadders/snakes/ISquare/isFirstSquare()|>,
  <|java+method://P2SnakesLadders/snakes/Square/landHereOrGoHome()|,|java+method
     ://P2SnakesLadders/snakes/ISquare/landHereOrGoHome()|>
...},
@modifiers={
  <|java+constructor://P2SnakesLadders/snakes/Snake/Snake(int,snakes.Game,int)|,
     public()>,
  <|java+field://P2SnakesLadders/snakes/Game/size|,private()>,
  <|java+constructor://P2SnakesLadders/snakes/Ladder/Ladder(int,snakes.Game,int)
     |,public()>,
  <|java+method://P2SnakesLadders/snakes/Square/moveAndLand(int)|,public()>,
    <|java+method://P2SnakesLadders/snakes/Ladder/destination()|,protected()>
  ...},
  @declarations={
    <|java+class://P2SnakesLadders/snakes/LastSquare|,|project://P2SnakesLadders/
        src/snakes/LastSquare.java|(17,180,<3,0>,<13,1>)>,
    <|java+method://P2SnakesLadders/snakes/Square/landHereOrGoHome()|,|project://
        P2SnakesLadders/src/snakes/Square.java|(678,94,<37,1>,<39,2>)>,
    <|java+parameter://P2SnakesLadders/snakes/Game/play(snakes.Die)/die|,|project
        ://P2SnakesLadders/src/snakes/Game.java|(733,7,<35,18>,<35,25>)>,
    <|java+field://P2SnakesLadders/snakes/Game/winner|,|project://P2SnakesLadders/
        src/snakes/Game.java|(241,6,<12,16>,<12,22>)>,
    <|java+method://P2SnakesLadders/snakes/Game/invariant()|,|project://
        P2SnakesLadders/src/snakes/Game.java|(252,115,<14,1>,<18,2>)>
  ...}
]
```