M^3: a General Model for Code Analytics in Rascal

Bas Basten[†], Mark Hills[*], Paul Klint[†], Davy Landman[†], Ashim Shahi[†], Michael Steindorfer[†], Jurgen Vinju[†]
[†]Centrum Wiskunde & Informatica, Amsterdam, The Netherlands
{Bas.Basten,Paul.Klint,Davy.Landman,Ashim.Shahi,Michael.Steindorfer,Jurgen.Vinju}@cwi.nl
[*]East Carolina University, Greenville, NC, USA
mhills@cs.ecu.edu

***Abstract*—This short paper introduces M3, a simple and extensible model for capturing facts about source code for future analysis. M3 is a core part of the standard library of the Rascal meta programming language. We motivate it, position it to related work and detail the key design aspects.**

I. Motivation

In the context of the EU FP7 project "OSSMETER" we have developed an infrastructure for measuring source code. The goal of OSSMETER is to obtain insight into the quality of open-source projects from all possible perspectives, including product, process, and community.[1]

The challenges faced by our part of the design, which focuses on source code, are *variability*, *integration*, and *accuracy*. Variability is necessary to support the different languages, including dialects, we support, as well as the different metrics we will compute. Integration is necessary at a semantic level, where metrics are computed across programming language and domain boundaries and are combined for further analysis. Accuracy is a prerequisite for insightful analyses. When accuracy is lost in the early stage of fact extraction from source code, the downstream analyses may show interesting but nevertheless meaningless information.

The standard solution for variability and integration is to put an explicit reusable model (e.g., database, graph) in the center, such that model producers (parsers & extractors) can be decoupled from model consumers (metrics & visuals). Examples of such intermediate models are FAMIX [1], RSF [2], GXL [3], KDM [4], and ATerms [5]. A positive side-effect of intermediate models is that the data are pushed into a general form, which enables integrating information from different sources. We have been inspired by these examples.

This paper is a brief description of M^3, a set of code models which should be easy to construct, easy to extend to include language specifics, and easy to consume to produce metrics and other analyses.[2] M^3 consists of an abstract syntax tree (AST) layer (hierarchical) and a relational (flat) layer. The AST layer uses a standard interface, but is expected to be language specific, while the relational layer is more abstract in nature and made for reuse.

The main threat to validity of any intermediate model which is generic enough to be reusable between different programming languages is that accuracy may be lost. This is why the M^3 standard first prescribes precise language-specific AST models and then language agnostic relational models where it is possible to abstract. The reader should be aware that we do *not* intend to create a unified model for programming language semantics. Such a language independent model would be inaccurate (wrong), and deliver meaningless metrics. Instead we opt for a unified *form* for storing facts about programs. This means that all models will have a predictable shape, but we do not assume any reusability of metrics or visualizations between models that are produced by different front-ends. The design of M^3 gives accuracy a higher priority than reuse.

The context of M^3 is the Rascal meta-programming language [6], [7].[3] This is a domain specific language specifically designed to include primitives we need to model the syntax and semantics of any programming language, and to analyze and manipulate these models. Three essential design elements related to this paper are that Rascal has value semantics for all in-memory data, including sets and relations; it has support for URI literals, called "source locations"; and it has term rewriting and relational calculus primitives to deal with hierarchical and relational data, respectively. This includes generic traversals [8] and pattern matching primitives as well as relational operators such as transitive closure and comprehensions.

The differences between M^3 and the aforementioned models is that M^3 deals with purely *immutable*, typed data which can be directly produced, manipulated and analyzed using Rascal primitives. Two unique elements are the introduction of URI literals to identify source code artifacts in a language agnostic manner and support for fully structured type symbols.

II. Experience Report

We developed M^3 to satisfy the requirements of the OSSMETER project, but we have been using it for a broader range of studies related to software analytics. We have used M^3 to construct comparable models of Android API evolution from three sources: source code, jar files and online documentation.[4] Creating the same model out the three sources required writing three different fact extractors. The first is based on the Eclipse JDT, the second uses ASM,[5] and the last uses Rascal's HTML library model. The analysis reveals interesting differences between the models which may influence the conclusions of related work [9]. For example, the Android source code releases contain undocumented public classes and methods. Nevertheless, these are available after deployment and developers use them.

[1]http://www.ossmeter.org
[2]An earlier, shorter version of this paper was presented at BENEVOL 2014.
[3]http://www.rascal-mpl.org
[4]Thanks to the students of the UvA Software Evolution course 2013.
[5]http://asm.ow2.org/

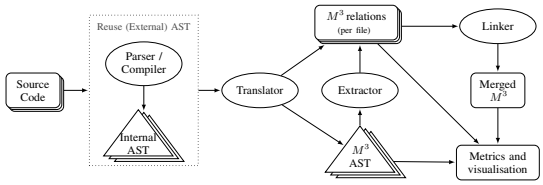

Fig. 1. An Overview: From Code to Metrics and Visuals via M^3 Models.

For OSSMETER, M^3 is applied to measure a plethora of object-oriented and procedural code metrics [10], [11] for Java and PHP. The metrics output between Java and PHP are comparable, and we also achieved some level of reuse when the metrics were based on the relational part of M^3, such as inheritance, overriding and containment relations.

We are using M^3 to improve our PHP static analysis framework called PHP AiR [12], supporting over-approximations of dynamic features such as file inclusion [13] and types.

We have used M^3 to analyze a large body of Java source code, comparing the cyclomatic complexity of methods to their size, refuting earlier scientific results reporting on a strong linear model between the two dimensions [14].

Finally a number of Universiteit van Amsterdam masters theses are based on using M^3 for PHP refactoring, PHP security analysis, quality metrics, concurrency-related Java refactorings, and Java reverse engineering.

III. Design aspects

A. Textual Models

M^3 is, like all Rascal data, fully typed and fully serializable as readable text with a standard notation that is equal to the expression syntax for literals. This means that any intermediate step can be visualized as plain text and not only searched and edited using standard text editing facilities, but also stored and retrieved persistently. One particular aspect of the Rascal IDE is that all printed source location literals (see below) in editors and consoles are treated as *hyperlinks*. M3 models are therefore "programmer friendly": easy to explore both interactively and programmatically using low-brow techniques.

B. Locations

To verify the correctness of metrics, or for explaining them, we want to trace measurements back to code. For example, when we present the largest class in a project, we need the size as well as a link to the source code of this class. In other words, we want to link information back to source code for all derived facts we produce. From the semantic web we take the idea of using URIs (Uniform Resource Identifiers), of the form `|<scheme>://<auth>/<path>?<qry>|(<off>,<len>)`, to model the identity of any artifact. We distinguish between two kinds of code locations: physical and logical. A *physical* location identifies a storage location, and may be absolute or relative. Examples of absolute physical locations are `|file:///tmp/Hello.java|` and `|http://foo.com/index.html|`, while an example of a relative location is `|project://MyPrj/Hello.java|`. It is always the *scheme* of a URI that defines to which root a URI is relative. In the case of `project`, it is an Eclipse project in the current workspace, while in the case of `file` it is the file system's root. The set of physical schemes is open and extensible, with schemes for Eclipse projects, Java class resources, OSGI bundle resources, JDBC data sources, jar files, and other resources.

TABLE I. Core Binary relations of M^3

Layer	**rel**[**loc**, **loc**]	Description
Generic core	`containment`	logical containment of code entities
	`declarations`	maps logical **loc** to physical **loc**
	`uses`	maps physical **loc** to logical **loc**
OO	`extends`	who extends who (logical **loc**)
	`implements`	who implements who (logical **loc**)
	`methodInvocation`	who invokes who (logical **loc**)
	`methodOverrides`	who overrides who (logical **loc**)
	`fieldAccess`	who accesses who (logical **loc**)

A *logical* code location is akin to a fully qualified name. For each specific language we design a naming scheme for each source code element that is, in some sense, declared. An example of a logical location is `|java+class://myProject/java/util/List|`. The scheme represents both the language and the kind of artifact that is identified. The URI *authority* declares the scope from which the name is resolved, in this case from `myProject` which depends on a particular version of the Java runtime. Finally, the path identifies the qualified name of the artifact in this scope. One goal of logical locations is to uniquely link to physical locations, at a certain moment in time, and at the same time be more or less stable under irrelevant code movement (such as moving the root source directory within a project). Another goal is for such links to be readable, writeable, recognizable and memorizable by human beings when developing new extractors, metrics or visuals. For instance, we might explore an M^3 model by projecting the information for an arbitrary class: the Rascal command `m@inheritance[|class://myPrj/java/util/List|]` would produce all interfaces that inherit from `java.util.List`.

M^3 models are built on this concept of logical and physical source locations. M^3 uses binary relations between locations; annotates AST nodes with these locations; and embeds these locations into symbolic facts (such as types) to link back to source code whenever possible.

C. Relations

The M^3 model is both layered and compositional. This means that M^3 models can be combined ("linked") and extended ("annotated"). The core relations are all between code locations (see Table I). An example containment tuple would be `<|class:///foo/Bar|,|pkg:///foo|>`.

This core model is language independent, facilitating not only volume metrics, browsing visuals (drill-down) and generic aggregation over containment relations, but also dependence between artifacts and thus impact and coupling/cohesion analyses. Also note that this core model is not restricted to handling programming languages. It can be used to model other kinds of formal languages like grammars, schema languages, or even pictorial languages.

For modeling language specific information we annotate the above core model with extra relations. Again these are binary relations between logical locations (Table I). These relations model key aspects of the static semantics of a programming language. Note that we never refer to instantiated or dynamic objects here, not even parametric type instantiations. All relations refer to source locations literally. For the accuracy of source code metrics, it is essential that M^3 separates what is written in the source code from what the code means dynamically. For example, if an abstract method from an interface is called we should not immediately infer all the call sites and add those to the invocations relation. Some metrics may want to count the fan-out to abstract methods, while other metrics want to know the impact on concrete implementations. You can compute such information by composing basic facts—e.g. "`invocation o overrides`" gives all the concrete callees for calls to abstract methods—and then compute a metric over the resulting relation.

D. Trees

For abstract syntax trees we use a general concept of algebraic data-types in Rascal. Every language comes with its own definitions. Algebraic data-types are easy to extend with new constructors (new programming language constructs). For M^3 we standardize some of the names used in defining AST types. In the core we standardize on five algebraic sorts to use when defining an abstract syntax: `Expression`, `Declaration`, `Statement`, `Type`, `Modifier`. The goal is to add as few extra sorts as possible when adding a new language. This leads to models which *over-approximate* the possible programs, but also increases the chance of reuse and extending existing fact extractors. For example, if all statements are in the same sort, then a basic function computing the cyclomatic complexity can be extended to cover a new language by just adding cases for the new types of statements (e.g. a `foreach` statement). We also provide annotation types for specific nodes, i.e. all nodes have a `src` annotation to point to the physical source location, all declarations may have a `decl` annotation to their logical location identifier and all Expressions may have a `type` annotation (see below). Trees are useful mostly for the computation of metrics over code units that contain statements, such as cyclomatic complexity, but also to infer data and control flow information for use in the more advanced analyses.

E. Types

For types we introduce a single sort called `TypeSymbol`. We use this to represent any kind of abstract value that variables and expressions in a language may produce. For Java we have a default set of type symbols to represent (parameterized) class and interface types, method signatures, and primitive types. These symbols can be used to compute with raw and parameterized types, either instantiated or uninstantiated. An example of a type symbol is: `class(|class:///java/util/List|, [class(|class:///java/lang/String|,[])])`, meaning the instantiated parameterized List type generated by the List class definition with its type parameter instantiated by the String class. We extended the core M^3 model with initial types—a relation from declarations to the types they generate—and we annotate the trees of expressions with the types they produce. Using type symbols we may compute with and reason about dynamic artifacts that are never declared yet may exist at run-time. For example, an upper-bound for the number of possible instantiations of a parameterized type can be computed based on such information.

F. Model Composition

When we extract M^3 models we do this incrementally, i.e. per file, per project, per composition of a project with its dependencies. Each file (in a given programming language) produces one M^3 model. Then the models for all files in a project are fused into one single M^3 model by applying set union to all the relations of the model. Finally, if there are project dependencies, we may fuse the M^3 models for different projects. Some analyses are best done before fusion. We compute the volume of a project before we fuse in the declarations of the jars we depend on. Other analyses are done only after fusing: depth of inheritance can only be computed if the models of classes we depend on are fully available. Since M^3 models are immutable values, like all Rascal values, such models can never be accidentally mixed. The `compose` function is called explicitly by the programmer to union the relation between two M^3 models; the `link` function does the same, but updates the authority fields of all logical locations such that uses in one project may point to declarations in another.

G. Efficiency and Memory Consumption

In essence a loaded M^3 model is an in-memory database of source code artifacts. For a large software system, the memory requirements are large and the efficiency is limited by I/O bottlenecks (network access, disk access and cache misses). Our current implementation still suffers from these effects, which are aggravated by value immutability. Since M^3 and Rascal are implemented on the JVM we are using "soft references" to store M^3 models, such that we can re-compute caches which have been garbage collected. The fused M^3 model and AST models for specific files often drop out of memory because of this, releasing necessary space. Soft references mitigate memory limitations nicely at the cost of runtime efficiency. On the other hand we are investing in low level optimizations for in-memory hierarchical data-structures [5], [15]. These results are hopeful, making it feasible to scale to even larger analyses without sacrificing immutability. Also scaling out in parallel will be facilitated by the immutability of M^3.

```
1 real estimateCoverage(M3 m) {
2   allCalls = m@methodInvocations + implicitCalls(m);
3   allCalls += allCalls o m@methodOverrides<to,from>;
4   testMethods = jUnit4TestMethods(m);
5   ifaceMethods = {met | <t, met> ←m@containment,
        isMethod(met), isInterface(t)};
6   reachable = {met | met ←reach(allCalls,
        testMethods), met in methods(m)} -
        allTestMethods - interfaceMethods;
7   total = methods(m) - testMethods - ifaceMethods;
8   return percentage(size(reachable), size(total),
        precision=0.01); }
```

Listing 1. Statically estimating test coverage using M^3.

H. Example

Listing 1 shows a concrete example of an M^3 analysis in Rascal. Function `estimateCoverage` analyzes any M3 model produced for Java and produces an over-estimate of a system's test coverage, given a JUnit4 test harness. The basic binary relations `methodInvocations`, `methodOverrides` and `containment` are used to find out who calls who. Relational operators such as infix `o` (composition), infix + (set union), infix – (set difference) and postfix + (transitive closure) are used to separate reachable from unreachable code. Comprehensions are used to generate and filter intermediate relations, and eventually a library function is called to compute the overall result, a percentage with given precision.

IV. Related Work

There is far too much related work to fit in this abstract. We only list primary inspirations for Rascal and M^3 here as an acknowledgment. Lexical techniques for fact extraction from source code use regular expression patterns to extract facts from program source code. These techniques are supported by languages such as Lex and AWK, Perl, or Python. Murphy and Notkin [16] extended the basic regular expression support provided by these tools and languages to include additional contextual information such that relations can be extracted once scanning is complete.

Approaches based on grammars naturally handle the nested constructs common to programming languages Parsing tools such as Yacc and ANTLR provide basic support for hand-coded fact extraction using the parser's semantic actions. A number of tools also exist with direct support for extracting facts from programs. Examples are the Rigi system [2], which provides fixed fact extractors for several languages and represents facts as tuples in the Rigi Standard Format (RSF). Meta programming languages, such as Stratego [17], TXL [18] and ASF+SDF [19] provide integrated parsing and tree processing support for source code fact extraction and manipulation in any programming language. SrcML [20] generates XML-annotated source code for the C language which can be queried directly using XML processors. Frameworks like JastAdd, Silver and Kiama make use of attribute grammars [21]–[25], using synthesized attributes to specify facts and inherited attributes to propagate these facts through the parse tree.

Relational approaches support extracting facts into relations which can then be combined and analyzed. Rigi relations are formed over RSF tuples and processed using operations in the Rigi Command Library (RCL). GROK [26] and CrocoPat [27] (using a notation called RML) instead use relational algebra; GROK supports only binary relations, while CrocoPat supports n-ary relations. The DEFACTO system [28], using RScript [29], also supports n-ary relations and relational algebra, as does Vankov's work on formulating program slicing using relational techniques [30].

Model-based approaches extract facts into models, representing abstractions of the systems being analyzed. MoDisco [31], [32], a model-based tool for modernizing legacy systems, supports extracting information about these systems into models defined using the Knowledge Discovery Metamodel [4], an OMG standard for modeling the structure and behavior of software systems. Moose [33], [34] extracts facts about software systems into models defined using the FAMIX [1] family of metamodels. Finally, we recommend reading about source code query mechanisms based on logic meta programming, such as Ekeko [35] and SOUL [36].

V. CONCLUSION

We have shown you a taste of M^3, an extensible and composable model for source code artifacts based on relations and trees, with immutable value semantics, source location literals, and extensibility with new types of binary relations. It has support for basic language independent analyses, and we have a detailed model for Java and PHP. M^3 is designed for variability, integration and accuracy, all required for software analytics research with a source code component.

The key open challenges for M^3 applications are efficiency and scalability, which we address using low level optimization techniques for hash-tries. Otherwise we are adding support for more programming languages. M^3 is open-source, EPL licensed (https://github.com/cwi-swat/rascal/library/analysis/m3).

REFERENCES

[1] S. Demeyer, S. Tichelaar, and S. Ducasse, "FAMIX 2.1—The FAMOOS Information Exchange Model," University of Bern, Tech. Rep., 2001.

[2] H. Müller and K. Klashinsky, "Rigi – A System for Programming-in-the-Large." in *Proceedings of ICSE'88*, April 1988, pp. 80–86.

[3] R. Holt, A. Winter, and A. Schürr, "GXL: Toward a Standard Exchange Format," in *Proceedings of WCRE'00*. IEEE, 2000, pp. 162–171.

[4] Object Management Group, "Knowledge Discovery Metamodel (KDM)," http://www.omg.org/technology/kdm/index.htm.

[5] M. van den Brand, H. de Jong, P. Klint, and P. Olivier, "Efficient Annotated Terms," *Software, Practice & Experience*, vol. 30, pp. 259–291, 2000.

[6] P. Klint, T. van der Storm, and J. Vinju, "EASY Meta-programming with Rascal," in *Post-Proceedings of GTTSE'09*, ser. LNCS. Springer, 2011, vol. 6491, pp. 222–289.

[7] P. Klint, T. van der Storm, and J. J. Vinju, "RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation," in *Proceedings of SCAM'09*. IEEE, 2009, pp. 168–177.

[8] M. van den Brand, P. Klint, and J. J. Vinju, "Term rewriting with traversal functions," *ACM Trans. Softw. Eng. Methodol.*, vol. 12, no. 2, pp. 152–190, 2003.

[9] T. McDonnell, B. Ray, and M. Kim, "An Empirical Study of API Stability and Adoption in the Android Ecosystem," in *Proceedings of ICSM'13*. IEEE, 2013, pp. 70–79.

[10] F. B. e Abreu, "The MOOD Metrics set," *ECOOP 95 Workshop on Metrics*, 1995.

[11] S. Chidamber and C. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[12] M. Hills and P. Klint, "PHP AiR: Analyzing PHP Systems with Rascal," in *Proceedings of CSMR-WCRE'14*. IEEE, 2014, pp. 454–457.

[13] M. Hills, P. Klint, and J. J. Vinju, "Static, Lightweight Includes Resolution for PHP," in *Proceedings of ASE'14*. ACM, 2014, pp. 503–514.

[14] D. Landman, A. Serebrenik, and J. Vinju, "Empirical Analysis of the Relationship between CC and SLOC in a Large Corpus of Java Methods," in *Proceedings of ICSME'14*. IEEE, 2014.

[15] M. J. Steindorfer and J. J. Vinju, "Code Specialization for Memory Efficient Hash Tries (Short Paper)," in *Proceedings of GPCE'14*. ACM, 2014.

[16] G. C. Murphy and D. Notkin, "Lightweight Lexical Source Model Extraction," *ACM TOSEM*, vol. 5, no. 3, pp. 262–292, 1996.

[17] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser, "Stratego/XT 0.16. Components for transformation systems," in *Proceedings of PEPM'06*. ACM Press, 2006, pp. 95–99.

[18] J. R. Cordy, "The TXL Source Transformation Language," *Science of Computer Programming*, vol. 61, no. 3, pp. 190–210, 2006.

[19] M. 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 *Proceedings of CC'01*, ser. LNCS, vol. 2027. Springer, 2001, pp. 365–370.

[20] J. I. Maletic, M. Collard, and H. Kagdi, "Leveraging XML Technologies in Developing Program Analysis Tools," in *Proceedings of the ICSE Workshop on Adoption-Centric Software Engineering (ACSE)*, 2004, pp. 80–85.

[21] M. Jourdan, D. Parigot, C. Julié, O. Durin, and C. L. Bellec, "Design, Implementation and Evaluation of the FNC-2 Attribute Grammar System," in *Proceedings of PLDI'90*, 1990, pp. 209–222.

[22] J. Paakki, "Attribute grammar paradigms - a high-level methodology in language implementation," *ACM Computing Surveys*, vol. 27, no. 2, pp. 196–255, June 1995.

[23] T. Ekman and G. Hedin, "The JastAdd system - modular extensible compiler construction," *Science of Computer Programming*, vol. 69, no. 1–3, pp. 14–26, 2007.

[24] A. M. Sloane, "Lightweight Language Processing in Kiama," in *Post-Proceedings of GTTSE'09*, ser. LNCS. Springer, 2011, vol. 6491, pp. 408–425.

[25] E. V. Wyk, D. Bodin, J. Gao, and L. Krishnan, "Silver: An extensible attribute grammar system," *Science of Computer Programming*, vol. 75, no. 1-2, pp. 39–54, 2010.

[26] R. Holt, "Binary Relational Algebra Applied to Software Architecture," University of Toronto, CSRI 345, March 1996.

[27] D. Beyer, A. Noack, and C. Lewerentz, "Simple and Efficient Relational Querying of Software Structures," in *Proceedings of WCRE'03*, 2003, pp. 216–225.

[28] H. J. S. Basten and P. Klint, "DeFacto: Language-Parametric Fact Extraction from Source Code," in *Proceedings of SLE'08*, ser. LNCS, vol. 5452. Springer, 2008, pp. 265–284.

[29] P. Klint, "Using Rscript for Software Analysis," in *Working Session on Query Technologies and Applications for Program Comprehension (QTAPC 2008)*, 2008.

[30] I. Vankov, "Relational approach to program slicing," Master's thesis, University of Amsterdam, 2005.

[31] H. Bruneliere, J. Cabot, G. Dupé, and F. Madiot, "MoDisco: A Model Driven Reverse Engineering Framework," *Information & Software Technology*, vol. 56, no. 8, pp. 1012–1032, 2014.

[32] H. Bruneliere, J. Cabot, F. Jouault, and F. Madiot, "MoDisco: A Generic and Extensible Framework for Model Driven Reverse Engineering," in *Proceedings of ASE'10*. ACM, 2010, pp. 173–174.

[33] O. Nierstrasz, S. Ducasse, and T. Gîrba, "The Story of Moose: An Agile Reengineering Environment," in *Proceedings of ESEC'05*, 2005.

[34] O. Nierstrasz, "Agile Software Assessment with Moose," *ACM SIGSOFT Software Engineering Notes*, vol. 37, no. 3, pp. 1–5, 2012.

[35] C. De Roover and K. Inoue, "The EKEKO/X Program Transformation Tool," in *Proceedings of SCAM'14*. IEEE, 2014, pp. 53–58.

[36] R. Wuyts *et al.*, "A logic meta-programming approach to support the co-evolution of object-oriented design and implementation," Ph.D. dissertation, PhD thesis, Vrije Universiteit Brussel, 2001.