Noname manuscript No.
(will be inserted by the editor)

ProMeTA: A Taxonomy for Program Metamodels in Program Reverse Engineering

Hironori Washizaki · Yann-Gaël Guéhéneuc · Foutse Khomh

Received: date / Accepted: date

Abstract To support program comprehension, maintenance, and evolution, metamodels are frequently used during program reverse engineering activities to describe and analyze constituents of a program and their relations. Reverse engineering tools often define their own metamodels according to the intended purposes and features. Although each metamodel has its own advantages, its limitations may be addressed by other metamodels. Existing works have evaluated and compared metamodels and tools, but none have considered all the possible characteristics and limitations to provide a comprehensive guideline for classifying, comparing, reusing, and extending program metamodels. To aid practitioners and researchers in classifying, comparing, reusing, and extending program metamodels and their corresponding reverse engineering tools according to the intended goals, we establish a conceptual framework with definitions of program metamodels and related concepts. We confirmed that any reverse engineering activity can be clearly described as a pattern based on the framework from the viewpoint of program metamodels. Then the framework is used to provide a comprehensive taxonomy, named Program Metamodel TAxonomy (ProMeTA), which incorporates newly identified characteristics into those stated in previous works, which were identified via a systematic literature review (SLR) on program metamodels, while keeping the orthogonality of the entire taxonomy. Additionally, we validate the taxonomy in terms of its orthogonality and usefulness through the classification of popular metamodels.

H. Washizaki
Dept. of Computer Science and Engineering, Waseda University / National Institute of Informatics / SYSTEM INFORMATION CO., LTD., Tokyo, Japan
Tel.: +81-352863272
Fax: +81-352863272
E-mail: washizaki@waseda.jp

Y.G. Guéhéneuc and F. Khomh
PTIDEJ–SWAT, DGIGL, Polytechnique Montréal, Quebec, Canada
E-mail: {yann-gael.gueheneuc, foutse.khomh}@polymtl.ca

Keywords Reverse engineering · Program metamodels · Program comprehension and analysis · Taxonomy

1 Introduction

Program reverse engineering plays an important role during software maintenance and evolution activities. This is because reliable information is often only embedded in the source code when maintaining and–or evolving a software system [21]. Program reverse engineering is the process of analyzing the program source code written in general purpose programming languages [43, 19]), to identify program code elements and create representations of a program at a certain level of abstraction.

Metamodels exist to describe and process software programs in program reverse engineering for program comprehension, maintenance, and evolution. They are essential for developing reverse engineering tools because they define constituents and relations to be identified in programs, enabling and circumscribing the features of the tools.

Reverse engineering tools often define their own metamodels according to their purposes and intended features [35]. The code representation (i.e., metamodel) depends on the actual reverse engineering problem and the aspired program analysis technique. Each reverse engineering tool must choose the appropriate abstraction level of the metamodel. For many reverse engineering activities, only a broad overview of the system is necessary. Consequently, the amount of extracted data by language analyzers (like compilers based on low-level metamodels) may become too large to comprehend or analyze in a reasonable amount of time [104]. On the other hand, some analysis requires details to ensure high precision and recall in the analysis results.

Each metamodel has advantages as well as limitations, which may be resolved by other metamodels. By conducting a systematic literature review (SLR) as a rigorous survey on program metamodels, we found that metamodels can be characterized by the following exhaustive orthogonal features: target language, abstraction level, meta-metamodel, exchange format, processing environment, definition, program metadata and history data, and quality.

Regarding the abstraction level, low-level metamodels represent the complete code syntax, high-level ones represent abstract architectural constituents, while mid-level ones represent neither the complete code syntax nor the architectural constituents [80]. Due to the differences between the metamodels, it is difficult to compare reverse engineering tools. These differences also lead to problems when exchanging information among tools [82]. For example, fact extractors often disagree and emit different facts for the same source program, undermining the users' understanding of the program and decreasing their confidence in the extractor [82]. For example, FAMOOS Information Exchange Model (FAMIX) [29] and Knowledge Discovery Metamodel (KDM) [90], which are popular metamodels, are applicable to the same programs, but have slightly different structures.

To assist practitioners and researchers in classifying, comparing, reusing, and extending program metamodels and the corresponding reverse engineering tools according to their goals, some works have evaluated and compared metamodels and tools [64, 61]. However, the comparisons and evaluations were conducted independently and do not provide a comprehensive guide of all the possible characteristics and limitations of metamodels.

The goal of this paper is to provide a comprehensive taxonomy and use this taxonomy to classify some popular metamodels. Our taxonomy, named Program Metamodel TAxonomy (ProMeTA), and the classification results support the classification, comparison, reuse, and extension of program metamodels and reverse engineering tools in various usage scenarios. To make the taxonomy and classification results consistent, we establish a conceptual framework with definitions of program metamodels and related concepts. The framework allows our taxonomy to incorporate newly identified characteristics into existing ones while keeping the orthogonality of the entire taxonomy.

We address the following research questions.

RQ1 Does ProMeTA cover all possible characteristics and limitations in existing works that evaluate and compare program metamodels?
RQ2 Does ProMeTA have orthogonality in its classification features?
RQ3 Is ProMeTA useful for guiding practitioners and researchers? Possible usecases include creating or choosing reverse engineering tools, and, communicating or researching program metamodels and reverse engineering tools.

This paper is an extended version of a paper presented at the 32nd International Conference on Software Maintenance and Evolution (ICSME) [121]. We have substantially added explanations on the taxonomy construction process and all of program metamodels found in the SLR. Moreover, we have added a reverse engineering pattern to make the conceptual framework and related terminology comprehensive. We summarize our contributions as follows:

– We developed a conceptual framework along with a pattern for program reverse engineering from the viewpoint of metamodels.
– Using a SLR to identify necessary features, we created a comprehensive taxonomy called ProMeTA. ProMeTA characterizes program metamodels in reverse engineering based on our framework.
– We classified existing popular program metamodels based on our taxonomy.

The remainder of this paper is organized as follows. Section 2 provides the background. Section 3 proposes our conceptual framework together with a program reverse engineering pattern, while we show the taxonomy construction process in Section 4. Section 5 shows our taxonomy. Section 6 validates and discusses our work. Finally, we provide our conclusion and future work in Section 7.

2 Background

2.1 Program reverse engineering

Reverse engineering is the process of analyzing a subject system in order to identify the system's constituents and create representations in other forms or at higher levels of abstraction [22]. Although reverse engineering can be initiated from any level of abstraction, this paper focuses on *program reverse engineering*, i.e., the process of analyzing a program source code to identify the program's constituents and create a representation of the program. This work is motivated by the fact that, when maintaining a software system, only the source code of the program often contains reliable information [21].

Moreover, this paper limits the target program codes to those written in general purpose programming languages (GPLs) [43], such as C and Java. Compared to domain specific languages (DSLs), which are used for a specific problem, GPLs are used to solve a broad spectrum of problems [19]. DSLs usually offer higher-level constructs (e.g., rules) in comparison to GPLs [66]. Thus, it is challenging for metamodels to describe GPL programs at appropriate abstraction levels according to specific purposes, such as program analysis [120], visualization [55], etc.

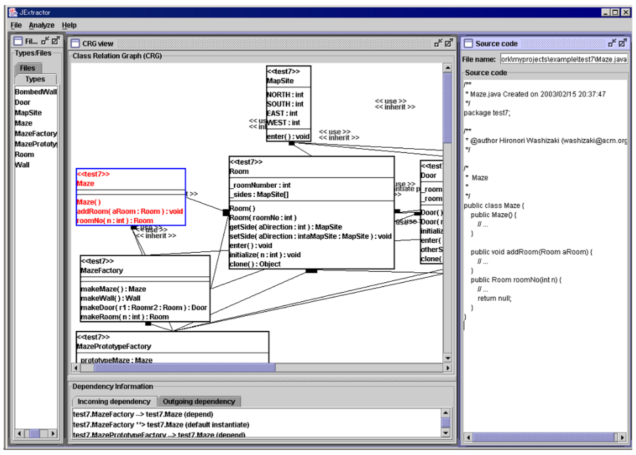
For example, the first author developed an automatic component-extraction system with its visualization (shown in Figure 1) targeting Java source code [120], which parses the Java source code, and selects only basic structural data such as classes, methods, fields, and dependencies among them with respect to a Java program metamodel.

Due to the above mentioned limitation, metamodels only handling domain specific languages (DLSs) such as SQL and XML are out of scope of this paper. Moreover the limitation leads to include metamodels and relevant reverse engineering approaches handling program source codes into the SLR, and exclude those handling only program bytecodes since bytecodes are not written in GPLs but machine-executable language specifications such as Java bytecode instructions[1].

2.2 Program Metamodel

Fact extraction from source codes aims to find pieces of information about a program (e.g., the name of a class or what function calls what function) [72]. Fact extraction is often the first step when analyzing a software system during reverse engineering. Before performing any high-level reverse engineering activity, the available information (i.e., facts) must be extracted and aggregated in a fact base or repository [72]. The *metamodel* (i.e., schema), which specifies

[1] Another reason to exclude those handling only bytecodes is because bytecodes often do not contain all the source code information. For example, Java bytecodes lack information on type parameters in generic types due to a mechanism called "Type Erasure".



```
/**
 * Maze.java Created on 2003/02/15 20:37:47
 */
package test7;

/**
 * @author Hironori Washizaki (washizaki@acm.org
 */

/*
 *  Maze
 *
 */
public class Maze {
   public Maze() {
      // ...
   }

   public void addRoom(Room aRoom) {
      // ...
   }
   public Room roomNo(int n) {
      // ...
      return null;
   }
}
```

```
test7.MazeFactory --> test7.Maze (depend)
test7.MazeFactory **> test7.Maze (default instantiate)
test7.MazePrototypeFactory --> test7.Maze (depend)
```

Fig. 1 Example of a view of our component-extraction system

the constituents and relations to be extracted, is essential to a fact extractor [72].

In addition, schemas are crucial to develop reverse engineering tools since they also specify the underlying semantic model of various analysis services [38]. From the viewpoint of modeling technology, herein schemas for fact extraction from programs are regarded as *program metamodels* while the extracted facts are regarded as models of the programs that conform to the corresponding schemas used for the extraction.

3 Terminology and Conceptual Framework

Although program metamodels are used under various contexts (e.g., forward engineering and reverse engineering) and at different abstraction levels from architecture to code, the concept of metamodels is not clearly defined. Indeed, there are many synonyms for "metamodel" including "schema", "representation", "format", and "form". Moreover, metamodels are often discussed along with standard exchange formats (SEFs) without a clear distinction between the two. For example, Sim and Koschke [104] report that the workshop focused on SEFs had a presentation addressing "a family of related SEFs including MOF, XMI, UML, XML and CDIF". However, a Meta Object Facility (MOF) [91] is a meta-metamodel, whereas the others are SEFs, although a UML can also serve as a program metamodel.

3.1 Terminology

To establish a common vocabulary, we first define the following core concepts:

- A *model* is a simplification of a system with an intended goal [39]. For example, a diagram showing only the program modules and their dependencies is a model of a program created with the goal of understanding the basic structure.
- A *metamodel* is a model of the language that captures the essential properties and features of a model [26]. In this context, a model is an abstraction of an aspect of the real world, while a metamodel is a further abstraction to describe the model. Although metamodels have primarily been developed and advertised by the Object Management Group (OMG) with its MOF standard [3] in the context of modelware, metamodels are not limited to MOF-based models. Examples of metamodels include *Program metamodels* in modelware, *schemas* (or *exchange format*) in dataware, and *grammars* in grammarware [39], which are models of program modeling languages, data languages, and programming languages, respectively in different technological spaces [76,123]. By referring to the ISO/IEC 42010:2011 terminology [58], a model is a "view" conforming to a "viewpoint" (i.e., metamodel) [18] [2].

3.2 Conceptual Framework

According to the above concepts, program metamodels and related concepts are defined below. Figure 2 shows the relationships among them following the OMG four-layer metamodel hierarchy [91,76] with some modifications to make it comparable with other model-driven engineering frameworks and views.

- A *program metamodel* is a model of a programming language grammar, which represents target programs according to a specific purpose. The elements of any program metamodel must be mapped to (a set of) elements of the corresponding grammar. As shown in Figure 2, "Program metamodel" is mapped by "Grammar". A program model must conform to its program metamodel. In Figure 2, "Program model" conforms to "Program metamodel". Examples include KDM, FAMIX, and UML.
- A program *metalanguage* is a language to describe program metamodels. In Figure 2, "Program metamodel" conforms to "Metalanguage". Metalanguages can be classified as *metasyntaxes of grammar* such as Extended BNF (EBNF) [56] in textual presentation or *meta-metamodels* of metamodels at certain abstraction levels such as MOF and Eclipse Modeling Framework (EMF) meta model Ecore [112] usually in a graphic presentation[3].

[2] Usually a single viewpoint corresponds to a single metamodel. However, KDM is a multi-viewpoint metamodel because a KDM specification provides a set of viewpoints that define a set of metamodel elements.

[3] In some environments [5,86,95], a graphical (i.e., visual) syntax can be used to define meta-models. However, such graphical syntax is usually intended to define domain specific modeling languages, which are graphical DSLs.

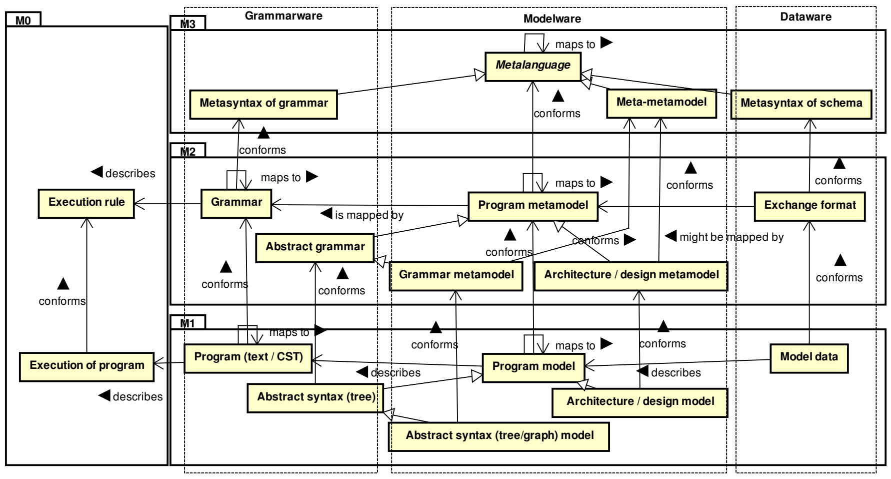


Fig. 2 Conceptual framework of program metamodels

– A *context-free grammar* (or simply *grammar*) is a formal device to specify which strings are part of the language, where the language is a set of strings over a finite set of symbols [33].
– A *concrete syntax tree (CST)* is a parse tree that pictorially shows how a string in a language is derived from the start symbol of the grammar [2].
– An *abstract syntax tree (AST),* is a simplified syntactic representation of the source code, excluding superficial distinctions of form and constituents that are unimportant for translation from the tree [2]. An AST follows an *abstract grammar*, which is a representation of the original concrete grammar at a higher-level of abstraction.
– An *abstract syntax model* is a graphical representation of an abstract syntax (tree). Abstract syntax models can be seen as low-level program metamodels. Examples include programming-language-independent AST models such as ASTM [89] [4] and programming-language-specific AST models such as Java Metamodel [73].
– A *standard exchange format (SEF)* (or simply an *exchange format*) is a metamodel (i.e., schema) of model data used to store the program models exchangeable among different tools (Figure 2). For example, "Model data" conforms to "Exchange format". Most of the elements in the exchange format can be mapped to (a set of) elements in the corresponding program metamodel. The exchange format may contain additional information (e.g., visual layout information) that is not included in the corresponding program model. Thus, "Exchange format" might be mapped by "Program metamodel" in Figure 2. Examples include XML, XML Metadata Interchange format (XMI), Resource Descriptor Format (RDF), Rigi Standard

[4] ASTM can also be a basis for deriving programming-language-specific AST models.

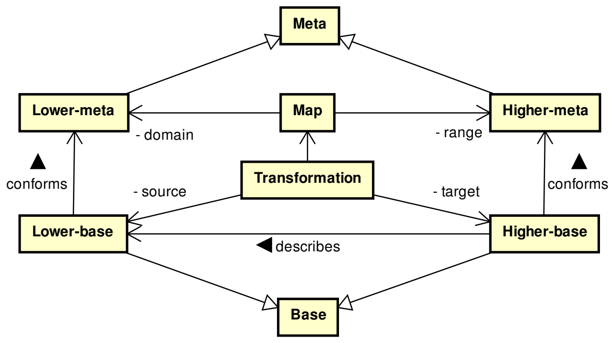


Fig. 3 Structure of **Transformation to higher abstraction levels**

Form (RSF), Tuple-Attribute Language (TA), GraX [104], CASE Data Interchange Format (CDIF) [54] and MSE [30]. Some of these (e.g., XMI and RDF) are general-purpose exchange formats that can be adapted to software, while others are specific to software [104].

3.3 Program Reverse Engineering as a Pattern

Based on the conceptual framework, now we can clearly explain any program reverse engineering activity and tool from the viewpoint of program metamodels. Program reverse engineering consists of various transformations such as extraction and abstraction.

Table 1 shows the pattern **Transformation to higher abstraction levels** that describes a common fundamental process of software transformation in any reverse engineering activity [122]. The pattern is described in the pattern form consisting of an *Alias* name (if necessary), a specific *Context*, a recurrent *Problem* under the context, its corresponding *Solution*, and a *Known implementation*.

The pattern gives specific reverse engineering activities (i.e., **Integrated program reverse engineering**, **Fact extraction** and **Architecture recovery** [122]) a common context, problem, and solution. By referring to this pattern, practitioners and researchers can recognize when, why, and how to perform reverse engineering together with underlying metamodels.

For example, by referring to the pattern, maintainers may develop a new tool or use an existing tool environment such as MOOSE [31] to comprehend Java source code. MOOSE can extract program entities and their relationships from Java source code by dealing with the abstract grammar of Java and FAMIX as a *Lower-meta* and a *Higher-meta*, respectively. MOOSE can visualize the extracted information in various forms such as the graph representation. Moreover, MOOSE can store the information in the form a compact serialization format called MSE.

Table 1 Pattern: Transformation to higher abstraction levels

Section	Description
Name	Transformation to higher abstraction levels
Context	You are analyzing software to comprehend or maintain it.
Problem	The description of the software contains too much data to be comprehended or analyzed in a reasonable amount of time. You have some interest in certain aspects on the software; however, its description is too complex to focus on particular aspects of the interest.
Solution	Transform the software (i.e., *Lower-base* in Figure 3) as a source to another as a target at a higher or the same level of abstraction (*Higher-base*). This is usually done by defining rules mapping from a metamodel at a lower level (i.e., *Lower-meta*) as the domain to another metamodel at a higher or the same level (i.e., *Higher-meta*) as the range. Figure 3 shows the elements involved in the transformation. Concrete transformations can be classified into four types: *Extraction*, *Abstraction*, *View* and *Store*. – *Extraction* transforms code artifacts based on a certain grammar to a set of program facts based on a certain program metamodel. It is usually done by a parser that parses code artifacts. – *Abstraction* transforms program models based on a certain lower metamodel to another model based on a certain higher metamodel. It is usually done by a filter component that queries, selects, and joins necessary data with respect to the higher metamodel; target higher metamodels are sometimes implicitly declared for the purpose of interactive ad hoc abstraction. – *View* transforms program models based on a metamodel to another model based on another visualization metamodel at a similar or almost the same abstraction level. The transformation results are then displayed. Typical examples are HTML tables, UML diagrams, and any general graph representation. – *Store* transforms program models based on a metamodel to model data according to an exchange format at a similar or almost the same abstraction level. Then the results are stored in a repository. Typical examples are XMI files, RDF files, and relational database.
Known implementation	Any reverse engineering tool.
Related patterns	The following patterns are based on combinations of multiple concrete transformations. – **Integrated program reverse engineering** performs *Extraction*, *Abstraction*, *Store*, and *View* in its solution. – **Fact extraction** performs *Extraction* and *Store* in its solution. – **Architecture recovery** performs *Extraction*, *Abstraction* and *View* in its solution.

4 Taxonomy Construction

Based on the background described in Section 2 and the vocabulary presented in Section 3, we identify various characteristics to distinguish existing program metamodels. We propose a comprehensive taxonomy, called the Program Metamodel TAxonomy (ProMeTA), to classify program metamodels in

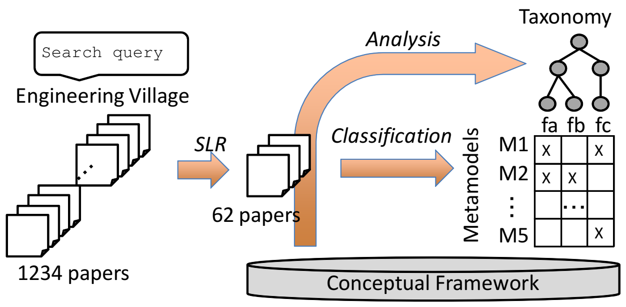


Fig. 4 Overview of taxonomy construction process

the form of feature diagrams based on our conceptual framework. ProMeTA integrates the characteristics stated in existing works with those newly identified, while maintaining the orthogonality of the entire taxonomy. Below its construction process is described in detail.

4.1 Construction Process

The development of a taxonomy can be approached in two different ways: top-down and bottom-up [117,45]. In the top-down approach, the taxonomy is built upon existing knowledge structures, allowing established definitions and categorizations to be reused, increasing the probability of achieving an objective classification procedure [117].

As previously mentioned, existing works have evaluated and compared program metamodels and tools, but none have provided a comprehensive guide that takes all possible characteristics into account. Therefore, we adopt a top-down approach to design ProMeTA based on our conceptual framework as follows. Figure 4 outlines the process.

1. A specific taxonomy is designed to accommodate a single, well-defined purpose, which is applicable to various circumstances [117]. First, we clearly define the specific purpose of ProMeTA – to support stakeholders in classifying, comparing, reusing, and extending program metamodels in program reverse engineering. Additionally, the taxonomy should support communications among stakeholders, improving the accessibility of the research results in program metamodels and reverse engineering.
2. Evidence-based Software Engineering (EBSE)[71] has been used to provide detailed insights regarding different topics in software engineering research and practice. A Systematic Literature Review (SLR) is known as the recommended EBSE method for aggregating evidence [70]. Using a SLR, existing works on classification and quality properties of program metamodels and tools are identified. Regarding the paper selection process within the SLR, we referred to the process adopted in another successful SLR [103]. The aim

of an SLR is to aggregate existing evidence on the research questions and to support the development of evidence-based guidelines for researchers and practitioners [70]. During the SLR, several popular metamodels are also identified.

3. Then, existing classifications, comparisons of program metamodels and related concepts [80, 64, 61, 13, 12, 79, 105, 42, 41, 10, 6] are analyzed. This information is merged into one structure in the form of feature diagrams [67] by referring to the basic term classification defined in our conceptual framework. Feature diagrams are trees that visualize the following relationships between a parent feature and its subfeatures (i.e., child features): "Mandatory", "Optional", "Or", and "Alternative". Mandatory means that subfeatures are required. Optional indicates subfeatuers do not have to be selected. Or implies at least one subfeature must be selected. Alternative denotes only one subfeature must be selected. A feature diagram essentially defines a taxonomy [27]. Additionally, the quality properties of program metamodels and related concepts discussed in papers [38, 76, 26, 105, 41, 115, 99, 62, 63, 27, 25, 124] as identified by the SLR are combined.
4. Then by referring to the basic term classification defined in the framework, all the identified characteristics in existing metamodels are added to the taxonomy while maintaining the orthogonality.
5. Finally, the taxonomy is validated in terms of its orthogonality, coverage, and usefulness by using it to classify the five popular metamodels identified in the SLR.

4.2 Systematic Literature Review

We searched for papers about program metamodels in reverse engineering using Engineering Village[5], which is a search platform providing access to 12 trusted engineering document databases, such as Ei Compendex and Inspec. The Engineering Village gives us the ability to search in all recognized scholarly engineering journals, conference, and workshop proceedings over different databases with a unique search query. Moreover the Engineering Village allows us to detect and remove most of duplicates in the search results automatically.

Because our main goal is to study characteristics of GPL-based program metamodels, and advantages and limitations that they offer to reverse engineering tools, metamodels dealing with program source code are regarded as study **subjects** while performing reverse engineering **tasks** processing program source codes as **stimuli**. Thus, we define the following three sets of keywords for defining our search query shown in Figure 5. In our search query, "*" at the end of each word is a truncation and can be replaced with zero or more characters.

– Subject: "meta model"[6] OR "meta models" OR metamodel* . We use this category to find papers that define and/or use a metamodel.

[5] http://www.engineeringvillage.com/

[6] It also finds papers having a keyword "meta-model".

```
(``meta model" OR ``meta models" OR metamodel*)
AND
(``source code" OR ``source codes" OR program*)
AND
(extract* OR transform* OR generat*)
```

Fig. 5 Search query

- Stimuli: "source code" OR "source codes" OR program* . We define this set to find studies based on the types of stimuli that are usually use in program metamodels studies.
- Task: extract* OR transform* OR generat* . These are simple yet sufficient to identify relevant papers since any reverse engineering objective and application must employ some sort of transformation; For example, extraction and generation can be regarded as types of vertical transformations [48].

4.3 Inclusion and Exclusion Criteria

We defined the following inclusion and exclusion criteria for the SLR. The relevance was verified by reviewing the title, the abstract, and if necessary, the body.

Inclusion criteria:

a). Studies published in journals or conference proceedings in the form of papers employing metamodels for program reverse engineering targeting program source code written in GPLs. For example, we included studies on program reengineering such as modernization and refactoring only if they employed program metamodels for the explicit reverse engineering phase as part of the entire reengineering process.

b). Studies that present details and–or complete results if a group reported more than one study on the same topic.

Exclusion criteria:

a). Studies that do not employ a program metamodel.

b). Studies that are not directly related to program reverse engineering targeting program source codes written in GPLs. For example, we exclude studies on model refactoring or transformation if they do not include any reverse engineering phase in the proposed refactoring or transformation process. We also exclude studies on parsers just for program compilation even though these parsers such as javac and the Eclipse Java Development Tools (JDT) in Java have internal metamodels [51]. For the same reason, we exclude studies just focusing on program transformations such as [16]. None of these implementations does provide an integration with standard metamodelling tools [51]; i.e., these are not originally intended for program reverse engineering objectives.

c). Elements of "grey" literature that are not published by trusted, well-known publishers, and do not use a well-defined referee process [20].
d). Articles not published in English.

4.4 Paper Selection Process

The process that we adopt to select the relevant papers is presented in Figure 6. In the figure, we present the set of activities that we undertook on the left while we present the number of remaining papers after each activity on the right as of October 13th 2015. In below, we explain each activity in detail.

a). Initial search: we execute our query in the Engineering Village. The search engine searches into the title, the abstract, and the keywords section of the papers looking for the keywords that are defined in our query to find the matching papers. The original set of papers provided by our proposed search query contains 1587 papers.
b). Automatic duplicate removal: we apply the Engineering Village's duplicate removal feature to automatically find and remove duplicate papers for first 1000 results[7].
1462 papers remain in the list.
c). Manual duplicate removal: we find and remove duplicates manually. Looking at the title, abstract and source (such as the conference name), we check whether the paper is duplicated or not. Although the Engineering Village search engine performed the duplicate removal process, there are still some duplicates in the result because different publishers use different formats to save and display the name of the authors, e.g., using initials instead of full names.
1234 papers remain in the list.
d). Apply inclusion and exclusion criteria: we perform this activity to check whether the paper is relevant or not by using the criteria. For each selected paper, one author conducts the initial check, while another author confirms the result of the initial check. In case of disagreement, there is a discussion among all authors until agreement.
We apply the inclusion/exclusion criteria and reduce the number of papers to 50. We use the title, the abstract and the body of the papers to remove irrelevant papers such as model-driven development works without any program metamodels for reverse engineering. 51 non-English papers are removed. 11 proceedings and books are also removed, because they list all of the accepted papers or chapters; we have already selected those related to the study.
e). Full analysis: we check whether there are multiple papers on the same approach reported by authors belonging a same group; in that case we

[7] The Engineering Village performs duplicate removal only for first 1000 results.

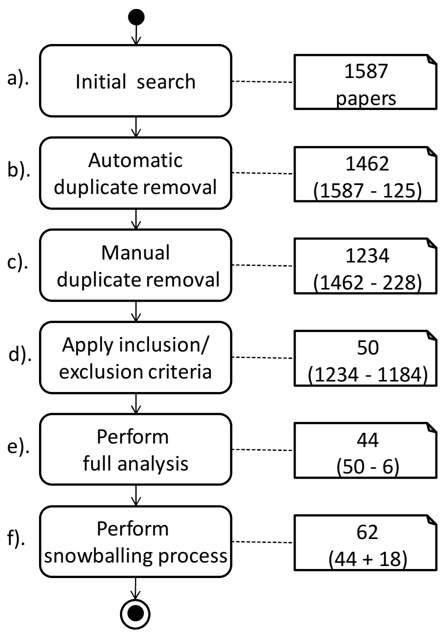


Fig. 6 The selection process with numbers showing number of papers remaining after each activity

include only a paper that represents most details. Moreover, we replace work summary papers such as a summary of a Ph.D. work with their complete version papers.
We perform the full analysis and remove 6 more papers from the list, leaving 44 papers [34, 14, 87, 23, 61, 83, 94, 60, 107, 8, 118, 113, 80, 82, 72, 17, 116, 77, 97, 85, 114, 7, 47, 98, 84, 24, 51, 73, 40, 96, 100, 32, 60, 83, 11, 49, 50, 124, 46, 4, 93, 68, 1, 74, 108, 109]. We remove two papers because [23] presents details about the proposed reverse engineering approach employing the logic-based program representation that is common in those three papers. We also remove [59] because [61] is its corresponding complete journal paper. There are three papers whose text we could not find online.

f). Perform the focused-snowballing process: for each selected paper, one author is assigned to go through the list of all references in order to find additional papers about classifications and quality properties of program metamodels.
By performing the snowballing process, we additionally identified 9 papers about classification and comparisons of program metamodels and related concepts [64, 61, 13, 12, 105, 42, 41, 10, 6] and 11 about quality properties [38, 26, 76, 105, 41, 115, 99, 62, 63, 27, 25].

5 Program Metamodel TAxonomy (ProMeTA)

ProMeTA consists of nine features that represent major points of variation (Fig. 7). Each feature is described below. Some of the features are designed to include concrete artifacts such as concrete programming languages if those are well known and accepted.

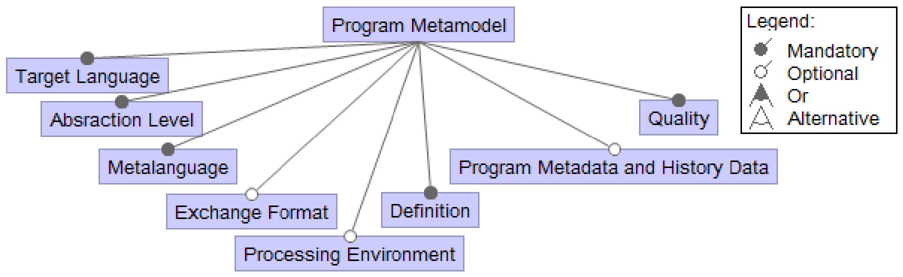


Fig. 7 Feature diagram for the program metamodels

5.1 Feature: Target Language

Language independence varies by metamodel; it depends on what kind of "Grammars" are supported and mapped by the "Program metamodel" as shown in Figure 2. Some metamodels only handle a certain language, while others handle multiple languages in a specific or any category; In later case, usually only common concepts among multiple languages are addressed. Even if a metamodel is stated to be "language independent", our analysis reveals that it often supports only a very limited number of languages. To define these characteristics precisely, the target language feature consists of two parts: language independence and current supported languages (Fig. 8)[8].

5.2 Feature: Abstraction Level

A representation (i.e., model) conforming to a metamodel must be as abstract as possible [75] within the limits of its reverse engineering objectives. Metamodels can be classified into three abstraction levels (Fig. 9): 1) low where the metamodel represents the complete syntax of a code, 2) high where the metamodel represents abstract architectural elements, and 3) middle where

[8] For simplification and comprehension, possible exclusion dependencies between subfeatures of Category Specific and subfeatures of Supported Language are omitted. For example, if Java is selected as the Supported Language and Category Specific is selected as the Language Independence, then Object-Oriented must be also selected as Category Specific. However, listing all such dependencies makes the taxonomy complex.

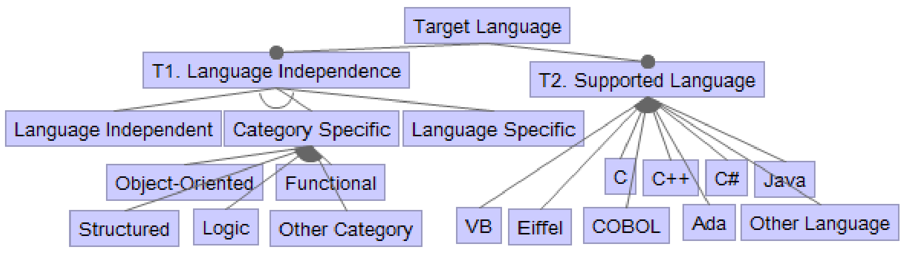


Fig. 8 Feature diagram for the target languages

the metamodel represents neither of the above [80]. In Figure 2, "Grammar metamodel" corresponds to 1), while "Architecture / design metamodel" corresponds to 2) and 3).

According to the requirements [79], SEFs should address classes (i.e., modules)[9], associations (i.e., relationships), and attributes. The same requirements can commonly be applied to high- or mid-level program metamodels; the domain ontology for integrating several reverse engineering tools (based on high- or mid-level metamodels) [64] specifies these characteristics. The ontology also contains other concepts such as System, Module (i.e., self-contained entity), SubProgram (i.e., non-self-contained entity), Variable, Containment relationship, and Use relationship [64], which are applicable to mid-level metamodels.

Regarding low-level metamodels, we follow the three representation aspects [42]: Lexical Structure, Syntax, and Semantics. Moreover, we add Dialects such as non-standard language specifiers as well as Preprocessor Artifacts [41] and Static/Dynamic semantics [6], taken from existing schema comparisons [41, 6]. For example, ASTM supports creating AST models for specific general purpose languages and DSLs as well as dialects and preprocessor artifacts of these languages.

5.3 Feature: Metalanguage

The data structures of SEFs used to represent software are classified as a Tree, a Graph, or Structured Data (i.e., data that is not a tree or a graph) [62]. We adopt the same classification for classifying metalanguages with our conceptual framework.

Based on the classification shown in Fig. 10, several well-accepted standard meta-metamodels together with the metasyntax of grammar, including MOF, EMF/Ecore, Kernel MetaMetaModel (KM3) [65], UML, and EBNF, are listed. In Figure 2, EBNF corresponds to "Metasyntax of grammar", while others correspond to "Meta-metamodel". KM3 is a meta-metamodel that has

[9] "Abstraction level" does not discriminate methods and classes from functions and modules since the former pair is applied on objects; "language independent" specifies whether metamodels cope with objects.

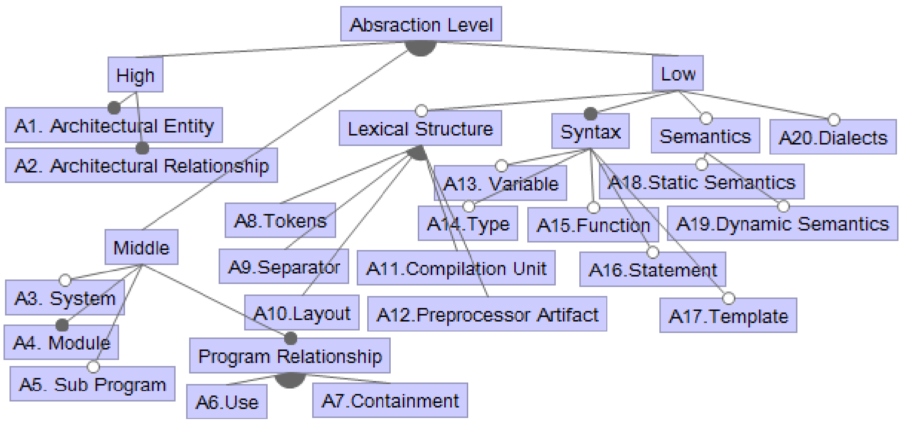


Fig. 9 Feature diagram for the abstraction levels

concepts similar to those found in MOF but is simpler than MOF [66]. Although UML is originally a modeling language classified in the M2 layer of the OMG four-layer metamodel hierarchy, it is often used to model program metamodels.

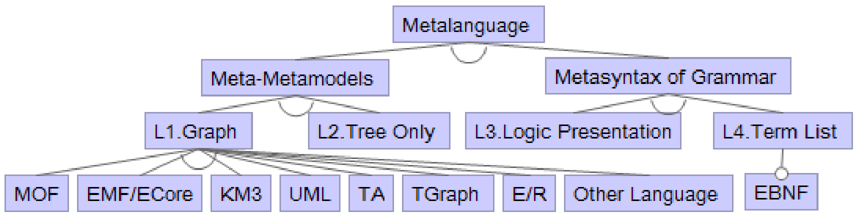


Fig. 10 Feature diagram for the metalanguages

5.4 Feature: Exchange Format

Program metamodels may depend on or have a high affinity with specific SEFs (i.e., "Exchange format" in Figure 2). However, it is preferable if program metamodels are independent from any SEF in order to exchange models among tools. For example, a reverse engineering tool environment called MOOSE [31] defines its own program metamodel called FAMIX, but it adopts CDIF (and later XMI and MSE) to exchange FAMIX-based information between different tools [88, 62].

Figure 11 shows the characteristic properties and considerations of SEFs [62, 63]. Among them, most quality characteristics, including scalability, simplicity, neutrality, formality, flexibility, evolvability, identity, solution reuse, and legibility, are examined according to the exchange patterns [63] (i.e., combinations of clarity and locality of the exchange format on which the metamodel depends).

The exchange format satisfies integrity only if a special mechanism to ensure an errorless exchange is provided [63]. If supported by many different tools, it satisfies popularity [63]. The exchange format satisfies completeness only if all the information in the metamodel can be included. On the other hand, the exchange format satisfies transparency only if no loss, alteration, or gain in the transferred information occurs due to the use of encoders and decoders [63].

As for the Abstract Syntax property, we list well-accepted SEFs, including Annotated Terms (ATerms) [15], InterMediate Language (IML), and Resource Graph (RG) [28], Multi-Layer, and Multi-Edge-Set (MLMES) graph [81], CASE Data Interchange Format (CDIF) [54], Tuple-Attribute Language (TA) [52], TA++ [79], and Datrix-TA [78], PROgramming with Graph Rewriting Systems (PROGRES) graph specification [102], GraX/TGraph [36], Graph Exchange Language (GXL) [53], Rigi Standard Form (RSF) [69], and MSE [30], along with general-purpose exchange formats, including XML [119] and XMI [92].

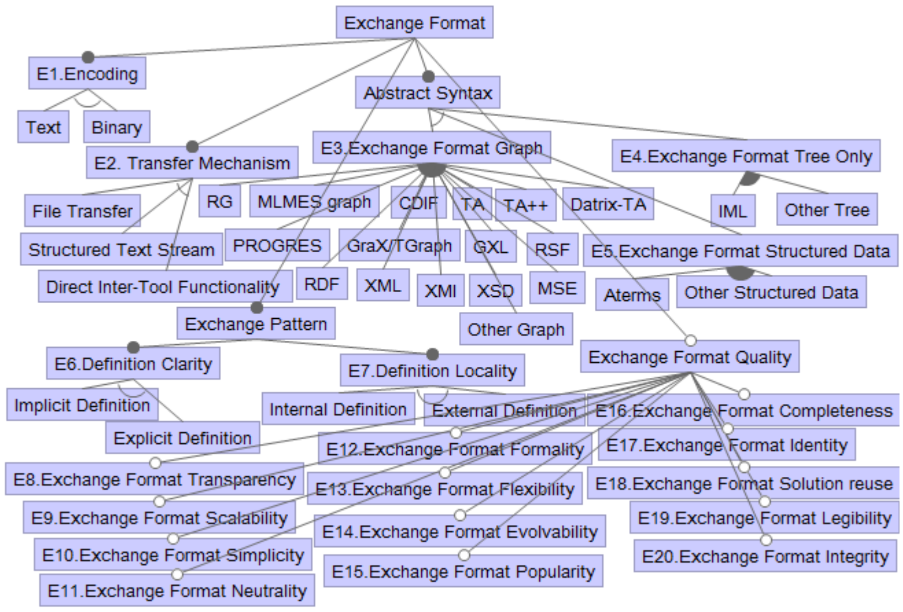

Exchange Format
E1.Encoding
Text
Binary
Abstract Syntax
E2. Transfer Mechanism
E3.Exchange Format Graph
E4.Exchange Format Tree Only
File Transfer
RG
MLMES graph
CDIF
TA
TA++
Datrix-TA
IML
Other Tree
Structured Text Stream
PROGRES
GraX/TGraph
GXL
RSF
E5.Exchange Format Structured Data
Direct Inter-Tool Functionality
RDF
XML
XMI
XSD
MSE
Aterms
Other Structured Data
Exchange Pattern
Other Graph
E6.Definition Clarity
E7.Definition Locality
Exchange Format Quality
Implicit Definition
Internal Definition
External Definition
E16.Exchange Format Completeness
Explicit Definition
E12.Exchange Format Formality
E17.Exchange Format Identity
E8.Exchange Format Transparency
E13.Exchange Format Flexibility
E18.Exchange Format Solution reuse
E9.Exchange Format Scalability
E14.Exchange Format Evolvability
E19.Exchange Format Legibility
E10.Exchange Format Simplicity
E15.Exchange Format Popularity
E20.Exchange Format Integrity
E11.Exchange Format Neutrality

Fig. 11 Feature diagram for the exchange formats

5.5 Feature: Processing Environment

By providing mechanisms to query (i.e., navigate) and transform program models, language toolkits, including reverse engineering tools, can fulfill analysis and comprehension tasks as well as maintenance and source code transformation tasks [9]. All these tasks follow the fundamental process of transformation as described in the **Transformation to higher abstraction levels pattern.** Specific processing environments to provide such mechanisms for navigation, transformation, analysis, and extraction are often provided together with the program metamodels. Figure 12 represents the major points of variations in the processing environment.

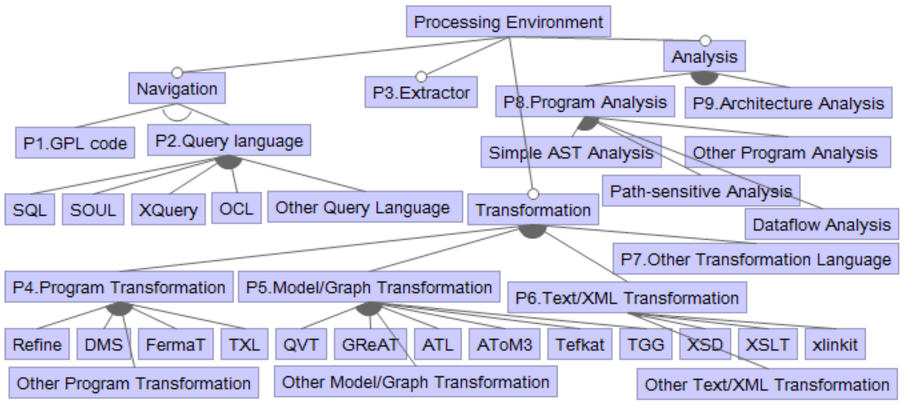


Fig. 12 Feature diagram for the processing environments

5.6 Feature: Definition

Typically, program metamodels are defined manually. Some approaches exist to automatically generate program metamodels from grammars [75,14], but they were originally intended for DSLs. Regarding the clarity and locality of the definitions, program metamodels can be classified into four exchange patterns similar to SEFs [62,63]: implicitly-internally defined, implicitly-externally defined, explicitly-internally defined, and explicitly-externally defined (Fig. 13).

5.7 Feature: Program Metadata and History Data

According to the requirements for SEFs [79], they should be able to store basic data (i.e., metadata) about the software systems they represent, including programming language versions, software system versions, file creation dates,

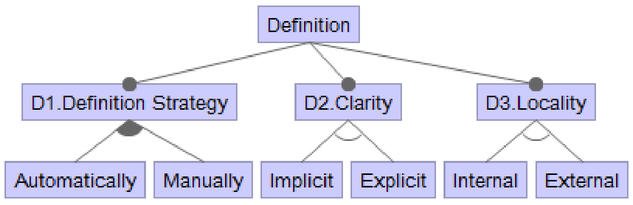


Fig. 13 Feature diagram for the definition

and file versions. We believe that program metamodels should handle such metadata together with the name of the programming languages.

Moreover, several program metamodels such as Ring [46] directly support the history data, allowing reverse engineering tools to work easily with source code versioning systems to conduct history analysis at some abstraction level. Figure 14 shows these characteristic properties.

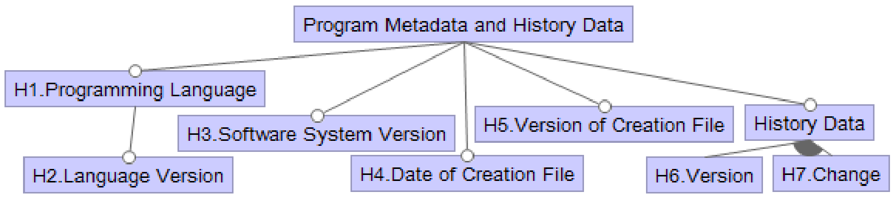


Fig. 14 Feature diagram for the program metadata and history data

5.8 Feature: Quality

We use the standard quality model ISO/IEC 25010:2011 [57] as the basis to specify the quality properties of the program metamodels in a comprehensive and consistent manner. During the SLR, we found 12 papers discussing quality properties that are applicable to program metamodels. They are requirements for SEFs [99,62,63], requirements for C++ schemas [41], requirements for reverse engineering tools enabled by schemas [38,115], comparative considerations for program comprehension tools [105], evaluation properties for static analysis frameworks [25], comparative issues for technological spaces [76], tracing features for model transformations [27], formality levels of metamodeling [26], and correctness of metamodels [124].

We categorized these properties along with those newly identified such as available form and verification into seven quality characteristics and their sub-characteristics defined in the ISO/IEC 25010:2011 quality model. Figure 15 shows the feature diagram for functional suitability, while Fig. 16 shows the feature diagram for the other quality characteristics. They can be summarized as follows:

- Functional suitability consists of three sub-characteristics: 1) functional appropriateness, which is mostly concerned with traceability [76, 27] from model elements to the corresponding portion of the source code, 2) functional correctness regarding how the program metamodel is verified [124], and 3) functional completeness regarding the applicability of the metamodel (i.e., general purpose metamodels or task-specific ones) [115]. In general, low-level metamodels are good for executability since any GPL should provide executable semantics, whereas most mid- or high-level metamodels lack executable semantics.
- Performance efficiency addresses the quantity of extracted data [105] and primarily depends on the granularity of the metamodel. A metamodel sacrifices such resource utilization if the ratio of the extracted information to code is very high.
- Compatibility addresses the interoperability among different tools and environments, which is broken down into several concrete properties. The identity (i.e., the identity preservation during transformation), solution reuse, and neutrality are primarily determined by the exchange patterns [63]. A metamodel satisfies integrity only if some special mechanism to ensure an errorless exchange has been provided with the metamodel [63]. A metamodel satisfies the instance representation [41] if a model can be easily represented in any SEF. This property is almost identical to the content-presentation separation [76].
- Usability addresses the learnability that is supported by the existence of documentations, samples, and user communities [25].
- Reliability addresses the availability of the program metamodel in terms of licensing [25]. Although metamodels should be fully available through websites or other means, sometimes only parts of a metamodel are provided.
- Maintainability encompasses five sub-characteristics. Among them, simplicity and evolvability are primarily determined by the exchange patterns [63]. Some metamodels have specific modularity mechanisms (such as packages) and–or reuse mechanisms (such as the inheritance and logical composition of metamodel elements) [27] to improve maintainability. The formality is specified as partially formalized or completely formalized [26] according to the available metamodel definition.
- Portability addresses adaptability and is composed of three concrete properties: flexibility and scalability are primarily determined by the exchange patterns [63]. A metamodel satisfies popularity if many different organizations beside the original developers have used it.

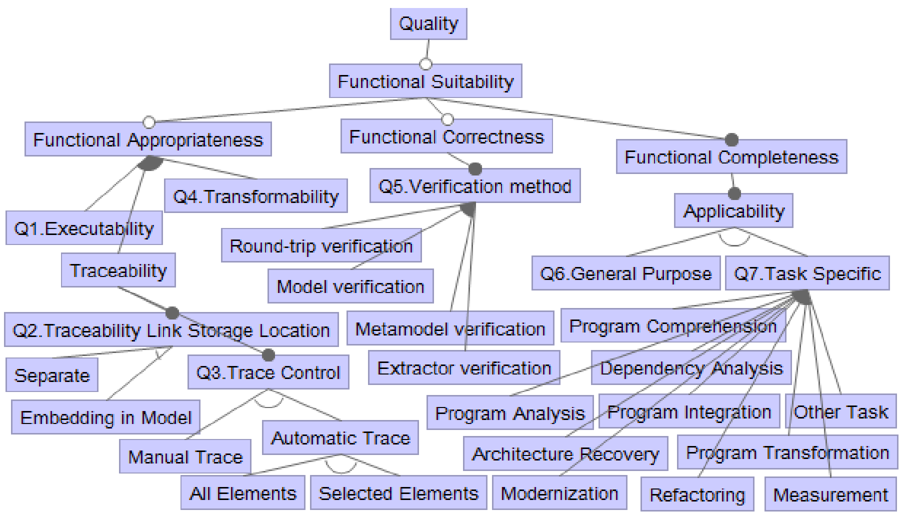


Fig. 15 Feature diagram for the functional suitability

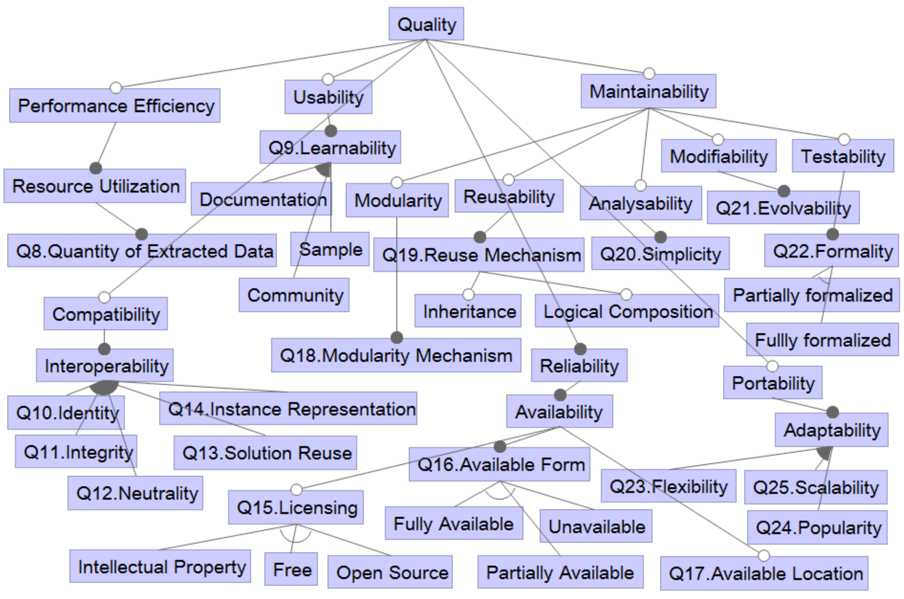


Fig. 16 Feature diagram for the performance efficiency, compatibility, usability, reliability, maintainability, and portability

6 Validation of ProMeTA

A taxonomy can be validated by demonstrating the orthogonality of its classification features, benchmarking against existing classification schemes, and demonstrating its utility to classify existing knowledge [106]. In our case, orthogonality means that a metamodel is classified as only one category of possible combinations of concrete features in the feature diagram. For example, the feature diagram of Definition yields 12 possible combinations of concrete features[10]. Each metamodel is classified into only one category such as (Manually, Implicit, Internal). We validated ProMeTA by classifying the popular metamodels identified in the SLR.

6.1 Target Popular Metamodels

For the concrete reverse engineering techniques or tools described in the set of 44 original papers obtained during the SLR, a total of 35 named and unnamed program metamodels were adopted[11]. Table 2 shows the list of the metamodels and corresponding papers. Surprisingly, most of papers adopted their own metamodels although there is not so much difference in characteristics and objectives. For example, there are seven similar AST-based metamodels (i.e., "Abstract Syntax *" in the table) defined independently. Moreover, there are four similar metamodels specific to the Java language (i.e., "Java * Model" in the table) but defined independently.

In Table 2, we identified that there are five program metamodels adopted in multiple papers[12]:

- M1. Abstract Syntax Tree Metamodel (ASTM): four papers [61,83,94,60]
- M2. Knowledge Discovery Meta-Model (KDM): five papers [96,100,32,60, 83]
- M3. FAMOOS Information Exchange Model (FAMIX): eight papers [17, 116,77,97,85,114,7,47]
- M4. SPOOL Metamodel: two papers [68,1] [13]
- M5. UNIQ-ART Metamodel: two papers [108,109] [14]

[10] These are the product of three subfeature sets: { Automatically, Manually, Automatically & Manually } × { Implicit, Explicit } × { Internal, External }.

[11] For illustrative purposes, we named the "unnamed" metamodels by a combination of concepts in our conceptual framework and metalanguages such as the "Abstract Syntax Tree Model in UML" in Table 2.

[12] This selection does not necessarily reflect actual adoption in program reverse engineering tools and projects.

[13] A more comprehensive paper [101] using the SPOOL Metamodel that is excluded from the SLR result.

[14] Some recent papers [110,111] using the UNIQ-ART Metamodel are not included in the SLR result.

Table 2 List of metamodels found in SLR

Metamodel	List of papers
Abstract Syntax Graph in TGraph	[34]
Abstract Syntax Metamodel in ECORE/EMF	[14]
Abstract Syntax Model in a graph grammar	[87]
Abstract Syntax Tree in logic representation	[23]
Abstract Syntax Tree Metamodel (ASTM)	[61,83,94,60]
Abstract Syntax Tree Model in MOF	[107]
Abstract Syntax Tree Model in UML	[8]
Architecture Model in TGraph	[34]
Columbus Schema	[118]
Common Meta-Model in common tree grammar	[113]
Daghstul Middle Metamodel	[80]
Datrix schema	[82]
Delphi metamodel in UML	[72]
Generic AST model in MOF	[98]
Grammar by EBNF	[14]
GXL schema in UML	[84]
Hismo	[47]
Integrated Meta-model of Reengineering in UML	[24]
JaMoPP Java Model	[51]
Java Meta Model in UML	[73]
Java MetaModel in grUML	[34]
Java Metamodel in MOF	[40]
KDM	[96,100,32,60,83]
FAMIX	[17,116,77,97,85,114,7,47]
MARPLE model in ECORE/EMF	[11]
Meta-model for design patterns and source code	[49]
Program entities and relationships in RDB	[50]
Program Metamodel in UML	[124]
Ring meta-model	[46]
Source Code Meta-Model in UML	[4]
SourcererDB Metamodel	[93]
SPOOL repository schema	[68,1]
System Engineering Technology Interface metamodel	[74]
UNIQ-ART Meta-model	[108,109]

6.2 Classification Results

We classified the aforementioned metamodels M1–M5 using ProMeTA (Fig. 17). The findings and corresponding suggestions for practitioners and researchers are summarized as follows:

– Target language: Of the five metamodels, three are language independent, while two handle object-oriented source code. Regardless of the language independence, all support the Java language since it seems to be the most common, especially in the context of reverse engineering research and practice. The second most common language is C++.
  If the target language is a major one like Java or C++, existing program metamodels and their corresponding reverse engineering tools may be reused, but if the target language is a minor one, a specific metamodel must be selected or a new one must be created.

- Abstraction level: All of the five metamodels can be used as mid-level metamodels, but only one metamodel (M2) can be used as a high-level one. According to the coverage of the low-level metamodel features, M1 and M2 are more useful even though they still miss some lexical structure features such as Token, Separator, and Layout. There are limited supports for language dialects.
  Practitioners and researchers can choose an appropriate metamodel and its corresponding reverse engineering tool according to their abstraction level requirements. However, our classification results indicate that none of the existing metamodels supports all of the required features at certain abstraction levels; in this case, it may be necessary to extend existing metamodels or create new one to cover the missing features.
- Metalanguage: Four of the five metamodels adopt the standard meta-metamodel MOF or the unified language UML, which are explicitly and externally defined, while only M5 adopts a specific implicitly-internally definition.
  If practitioners and researchers adopt various tools for long-term usage, it may be better to choose or create program metamodels (like M1–M4) defined by widely accepted, explicitly-externally defined metalanguages (especially MOF and UML).
  In addition, the existence of user communities of metamodels could contribute to the ease of usage of their metalanguages; for example, since M3 has a large user community as identified regarding the feature Q9: Learnability, its metalanguage UML could be a good choice for creating (or selecting) program metamodels.
- Exchange format: Corresponding to the metalanguage used, three of the five metamodels adopt standard SEFs such as XMI, which are explicitly-externally defined, while M5 supports a specific binary-based implicitly-internally defined data exchange.
  If practitioners and researchers consider utilizing various tools for long-term usage, selecting or creating program metamodels with a good exchange format quality (like M1, M2 and M4), which support the widely accepted, explicitly-externally defined SEFs (especially XMI) may be a better choice; however, its impact on selection or creation could be less than those of other features (such as the abstraction level) since specific exchange formats can be additionally supported by preparing convertors among exchange formats, unless the metamodel originally supports explicitly-externally defined SEFs.
- Processing environment: Due to their popularity, all of the five metamodels have dedicated extractors and navigation supports. It is obvious that extractors and navigation supports should be prepared to improve the ease of use of any program metamodels.
  There are dedicated transformation supports including refactoring facilities for three of five. Most of the metamodels (except for M5) are suitable for transformations and program analysis. Practitioners and researchers should

check whether the processing environment and facilities are available to meet their reverse engineering objectives.

- Definition: All of the five metamodels are manually defined. All except M5 are explicitly defined, leading to high quality metamodels with high compatibility, maintainability, and portability. Three of which are externally and fully formalized. The other two (M4 and M5) are internally defined.
  If practitioners and researchers utilize various tools for long-term usage, selecting or creating explicitly-externally defined metamodels (like M1–M3) is a better choice.
- Program metadata and history data: There are few supports to describe meta and history data in metamodels; only the programming language name and the file version are supported by M1 and M2, respectively [15].
  During the SLR, several history-aware metamodels were found to explicitly address the version history: Ring [46], Hismo [44,47], FAMIX-based RHDB code model [7] and FAMIX-based ArchEvoDB schema [97]. If practitioners and researchers conduct reverse engineering in which history analysis is taken into account, selecting a history-aware metamodel, especially the RHDB code model and the ArchEvoDB schema, may be better since these are defined as extensions of FAMIX, which is a widely accepted popular metamodel.
- Functionality: Two metamodels (M1 and M2) support most of the functional suitability features, including executability, traceability, and transformability, since these are low-level metamodels supporting static and dynamic semantics shown in the abstraction level features. None explicitly state how these have been verified. Although most can be used for various purposes, only M5 is for several specific tasks such as the dependency analysis.
  Practitioners and researchers should verify whether the potential program metamodels satisfy their reverse engineering functionality requirements. If a metamodel is used for various reverse engineering purposes, selecting a general one (like M1–M4) is better.
- Non-functionality: Only M1 sacrifices the performance efficiency since it contains all of the statement-level code descriptions. Three (M1–M3) have a good usability since documents and samples with communities are well prepared. These three metamodels also have good compatibility, maintainability, and portability since these are explicitly-externally defined, fully formalized, and fully available. Unfortunately the definitions of M4 and M5 seem to be unavailable elsewhere on the Internet or in the literature. Most of the metamodels (except M5) support inheritance and logical composition as reuse mechanism. However, only M2 supports the dedicated modularity mechanism.
  Practitioners and researchers should check whether potential program metamodels satisfy their non-functionality requirements. If existing metamodels

[15] Some of the metamodels (especially M1) can be extended to include metadata and history data.

are to be reused, they must select fully available and formalized metamodels (like M1–M3).

The above-mentioned findings and suggestions can be summarized as follows. Existing program metamodels can be reused for major languages such as Java and C++. It is better to choose and/or create program metamodels defined by explicitly-externally defined major metalanguages. It is better to choose and/or create program metamodels associated with explicitly-externally defined SEFs. Most of popular program metamodels are suitable for transformations and program analysis; however, few support to describe meta and history data.

M	Target Language		High		Middle					Lexical Structure					Syntax					Semantics		Dialects
	T1	T2	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13	A14	A15	A16	A17	A18	A19	A20
M1	Independent	Java, Delphi			X	X	X	X	X				X	X	X	X	X	X	X	X	X	X
M2	Independent	Java, PL/SQL	X	X	X	X		X	X				X	X	X	X	X		X	X	X	
M3	Object-Oriented	Java, C++, Ada, Smalltalk			X	X		X							X	X	X					
M4	Object-Oriented	Java, C++			X	X		X								X	X		X			
M5	Independent	Java, C++, C			X	X		X							X	X	X			X	X	

M	Meta-Language				Exchange Format																					
	L1	L2	L3	L4	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10	E11	E12	E13	E14	E15	E16	E17	E18	E19	E20	F21	F22
M1	MOF				Text	File Transfer	XMI, XSD			Exp	Ext	++	++	+	++	++	++	++	+	+	++	+	+		Exp	Ext
M2	MOF				Text	File Transfer	XMI			Exp	Ext		+	+	++	++	++	++	+	+	++	+	+		Exp	Ext
M3	UML				Text	Text Stream	MSE			Exp	Ext		+	+	++	++	++	++	+	+	++	+	+		Exp	Ext
M4	UML				Text	File Transfer	XMI			Exp	Ext		+	+	++	++	++	++			++	+	+		Exp	Ext
M5				X	Binary	Direct			RDB	Imp	Int		–	–	–	–	–	–			–	–	–		Imp	Int

M	Processing Environment								
	P1	P2	P3	P4	P5	P6	P7	P8	P9
M1		OCL, Modisco Java Model Query, Knowledge Discovery Metamodel SDK	MoDisco (dedicated parsers), Gra2MoL		ATL, MoDisco, ADM tools			X	
M2		OCL, Modisco Java Model Query, Knowledge Discovery Metamodel SDK	MoDisco (KDM Source Discovery, Java Discoverer)		ATL, MoDisco, ADM tools			X	X
M3		MOOSE Navigation and Querying Engine	MOOSE		MOOSE Refactoring Engine			X	X
M4	X		Datrix				X	X	
M5		SQL	SPOOL (dedicated extractors)						X

M	Definition			Program Metadata and History Data							Functionality						
	D1	D2	D3	H1	H2	H3	H4	H5	H6	H7	Q1	Q2	Q3	Q4	Q5	Q6	Q7
M1	Manually	Exp	Ext	X							+	Embedded	Manual	+		+	
M2	Manually	Exp	Ext						X		+	Embedded	Manual	+		+	
M3	Manually	Exp	Ext													+	
M4	Manually	Exp	Int											+		+	
M5	Manually	Imp	Int														Program Comprehension, Dependency Analysis, Program Analysis, Architecture Recovery

M	Non-Functionality																
	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q18	Q19	Q20	Q21	Q22	Q23	Q24	Q25
M1	–	Doc, Sample, Community	++		++	+	+	Free	Fully		Inheritance, Composition	+	++	Fully	++	+	++
M2		Doc, Sample, Community	++		++	+	+	Free	Fully	Package	Inheritance, Composition	+	++	Fully	++	++	+
M3		Doc, Sample, Community	++		++	+		Free	Fully		Inheritance, Composition	+	++	Fully	++	++	+
M4			++		–	–		Free	Unavailable		Inheritance, Composition	+	–	Partially	+		+
M5			–		–	–		Free	Unavailable			–	–	Partially	–		–

Fig. 17 Classification results using ProMeTA (M1: ASTM, M2: KDM, M3: FAMIX, M4: SPOOL Metamodel, M5: UNIQ-ART Metamodel, X: supports the characteristic indicated, ++: particularly satisfies the characteristic/requirement indicated, +: satisfies the characteristic/requirement indicated, -: sacrifices or does not satisfy the characteristic/requirement indicated, Exp: Explicit, Imp: Implicit, Ext: External, Int: Internal)

6.3 Discussions

RQ1: Does ProMeTA cover all possible characteristics and limitations in existing works that evaluate and compare program metamodels?

During the construction process of ProMeTA, the important characteristics from existing classification schemes/frameworks and comparisons [80,64,61, 13,12,79,105,42,41,10,6] as well as discussions on quality properties [38,26,76, 105,41,115,99,62,63,27,25,124] for program metamodels and related concepts identified by the SLR were included or mapped to the items, implying that it has adequate coverage. Thus, ProMeTA is implicitly benchmarked against existing classification schemes.

RQ2: Does ProMeTA have orthogonality in its classification features?

We successfully classified popular program metamodels from the SLR according to the characteristics defined in ProMeTA and show how it can help classify program metamodels. Moreover, the classification did not result in the characteristics fitting into more than one category, demonstrating the orthogonality of the classification features.

RQ3: Is ProMeTA useful for guiding practitioners and researchers?

ProMeTA can guide practitioners and researchers in the following possible usecases UC1–UC3.

- UC1. Developing new reverse engineering tools: When engineers want to build their own reverse engineering tools, they must define the requirements in program metamodels that enable and circumscribe the features of the tools. ProMeTA supports the requirements definition and guides reuse, extension, or creation of metamodels because engineers can recognize features included in ProMeTA as possible requirement items. Moreover, if a ProMeTA-based classification result of a potential metamodel for reuse or extension is available like M1–M5 in the above validation, engineers can easily determine whether the metamodel satisfies their requirements.
- UC2. Choosing existing reverse engineering tools: When engineers want to reuse and eventually extend existing reverse engineering tools, they must compare and then select the appropriate one according to how the underlying program metamodels meet their objectives. ProMeTA can help by comparing criteria and the metamodels according to the characteristics defined in ProMeTA. Moreover, ProMeTA may help by comparing existing classification results of metamodels (if available).
- UC3. Communicating or researching program metamodels and reverse engineering tools: ProMeTA can serve as a reference for the reverse engineering community, including practitioners and researchers. It can be extended by peers, providing the community with an important body of knowledge to guide future communications and research on program metamodels and the corresponding reverse engineering tools since it incorporates the characteristics of metamodels into a single orthogonal structure based on a concep-

tual framework that defines common terminology. For example, ProMeTA can serve as the basis for building an open repository of information of existing program metamodels (and corresponding tools) by accumulating classification results. The above-mentioned classification results of M1–M5 can be a starting point.

6.4 Limitations

Five popular metamodels are identified solely on their adoption in papers selected by the SLR. It is plausible that such "popularity" does not reflect actual popularity in program reverse engineering tools and projects. In the future, we will investigate actual adoptions of metamodels in active projects on reverse engineering tools and classify these metamodels using ProMeTA.

The classification of the five popular metamodels based on ProMeTA was conducted by the first author of this paper and reviewed by the second and third authors. Therefore, it is possible that our classification results may not be completely correct. To mitigate this threat to validity, we have opened the classification results and ProMeTA to the public and call for comments at our Website[16]. We also contacted the original developers of the metamodels addressed in the paper. Their comments are incorporated into our classification[17].

We used Engineering Village as the initial document base of the SLR. Although it is adopted in other SLRs [103], relevant papers may be missed. Additionally, we may have missed relevant papers even after double-checking the paper selection results. To mitigate these threats, we plan to use other databases, extend our SLR, and elicit public review of the revised results.

Although our rigorous systematic literature survey identified the characteristics of program metamodels, other characteristics to be used for classification of metamodels may be omitted. ProMeTA is expected to efficiently incorporate such missing characteristics into the single structure because the form of feature diagrams should make such an extension of the taxonomy easy.

Any taxonomy can only unleash its full potential through widespread awareness and a large number of contributions [37]. Therefore, our future work is to follow a popularization strategy [37].

7 Conclusion and Future Work

In this paper, we propose a conceptual framework with definitions of program metamodels and related concepts as well as build a comprehensive taxonomy named ProMeTA based on this framework. ProMeTA incorporates newly identified characteristics into those stated in existing works via a systematic

[16] http://www.washi.cs.waseda.ac.jp/prometa/
[17] As of 22nd September 2017, we have received and incorporated comments provided by the original developers of M1, M2, M4 and M5.

literature survey on program metamodels, while maintaining the orthogonality of the entire taxonomy. This feat is accomplished by referring to the basic term classification defined in the framework. Additionally, we validate the taxonomy in terms of its orthogonality and usefulness through the classification of five popular metamodels from the survey. We have made ProMeTA available to the reverse engineering community, including practitioners and researchers, through our Website.

In the near future, we plan to validate ProMeTA by conducting experiments involving the three usecases (UC1–UC3) in Section 6. This should provide improved answers to the research questions, especially RQ3. We are also planning a collaborative Wiki to let the community refine or modify ProMeTA online.

Over the long term, we plan to extend our SLR using additional databases and share the revised results to obtain reviews from the public. We expect that the research community will further validate ProMeTA as well as the SLR results from the viewpoints of practitioners and researchers. Public input should not only lead to standard terminology and classification characteristics in the taxonomy, but also extend the taxonomy to include new categories and datasets that reflect its usage.

Acknowledgements We thank reviewers as well as the original developers of the metamodels addressed in the paper for their valuable comments, which significantly improved our paper. This work was supported by JSPS KAKENHI Grant Number 16H02804, IISF SSR Forum 2015 and 2016.

References

1. Abdi, M.K., Lounis, H., Sahraoui, H.A.: Analyzing Change Impact in Object-Oriented Systems. In: Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA), pp. 310–319. IEEE Computer Society (2006)
2. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D. (eds.): Compilers: Principles, Techniques, and Tools (2nd Edition), 2 edn. Addison-Wesley Professional (2006)
3. Alanen, M., Porres, I.: A Relation Between Context-Free Grammars and Meta Object Facility Metamodels. TUCS Technical Report, No.606 pp. 1–13 (2003)
4. Alikacem, E.H., Sahraoui, H.A.: A Metric Extraction Framework Based on a High-Level Description Language. In: Proceedings of the 9th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 159–167. IEEE Computer Society (2009)
5. Amelunxen, C., Klar, F., Königs, A., Rötschke, T., Schürr, A.: Metamodel-based Tool Integration with MOFLON. In: Proceedings of the 30th International Conference on Software Engineering (ICSE), pp. 807–810. ACM (2008)
6. Amelunxen, C., Königs, A., Rötschke, T.: MOSL: Composing a Visual Language for a Metamodeling Framework. In: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), pp. 81–84. IEEE Computer Society (2006)
7. Antoniol, G., Penta, M.D., Gall, H.C., Pinzger, M.: Towards the Integration of Versioning Systems, Bug Reports and Source Code Meta-Models. Electronic Notes in Theoretical Computer Science **127**(3), 87–99 (2005)

8. Antoniol, G., Penta, M.D., Merlo, E.: YAAB (Yet Another AST Browser): Using OCL to Navigate ASTs. In: Proceedings of the 11th International Workshop on Program Comprehension (IWPC), pp. 13–22. IEEE Computer Society (2003)
9. Antoniol, G., Penta, M.D., Merlo, E.: YAAB (Yet Another AST Browser): Using OCL to Navigate ASTs. In: Proceedings of the 11th IEEE International Workshop on Program Comprehension (IWPC), pp. 13–22. IEEE Computer Society (2003)
10. Arcelli, F., Masiero, S., Raibulet, C., Tisato, F.: A Comparison of Reverse Engineering Tools based on Design Pattern Decomposition. In: Proceedings of the Australian Software Engineering Conference (ASWEC), pp. 262–269. IEEE Computer Society (2005)
11. Arcelli, F., Zanoni, M., Porrini, R., Vivanti, M.: A Model Proposal for Program Comprehension. In: Proceedings of the 16th International Conference on Distributed Multimedia Systems (DMS), pp. 23–28. Knowledge Systems Institute (2010)
12. Armstrong, M., Trudeau, C.: Evaluating Architectural Extractors. In: Proceedings of the 5th Working Conference on Reverse Engineering (WCRE), pp. 30–39. IEEE Computer Society (1998)
13. Bellay, B., Gall, H.: A Comparison of Four Reverse Engineering Tools. In: Proceedings of the 4th Working Conference on Reverse Engineering (WCRE), pp. 2–11. IEEE Computer Society (1997)
14. Bergmayr, A., Wimmer, M.: Generating Metamodels from Grammars by Chaining Translational and By-Example Techniques. Proceedings of the 1st International Workshop on Model-driven Engineering By Example co-located with ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems (MoDELS), CEUR Workshop Proceedings **1104**, 22–31 (2013)
15. van den Brand, M., Klint, P.: ATerms for Manipulation and Exchange of Structured Data: It's All about Sharing. Information and Software Technology **49**(1), 55–64 (2007)
16. Bravenboer, M., Kalleberg, K.T., Vermaasc, R., Visser, E.: Stratego/XT 0.17. A Language and Toolset for Program Transformation. Science of Computer Programming **72**, 52–70 (2008)
17. Brühlmann, A., Gîrba, T., Greevy, O., Nierstrasz, O.: Enriching Reverse Engineering with Annotations. In: Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS), pp. 660–674. Springer-Verlag (2008)
18. Brunelière, H., Cabot, J., Dupé, G., Madiot, F.: MoDisco: A Model Driven Reverse Engineering Framework. Information and Software Technology **56**(8), 1012–1032 (2014)
19. Buchner, T., Matthes, F.: Introspective Model-Driven Development. Proceedings of the 3rd European Workshop on Software Architecture (EWSA), Lecture Notes in Computer Science **4344**, 33–49 (2006)
20. Budgen, D., Burn, A., Brereton, O., Kitchenham, B., Pretorius, R.: Empirical Evidence about the UML: A Systematic Literature Review. Software: Practice and Experience **41**(4), 363–392 (2011)
21. Canfora, G., Penta, M.D., Cerulo, L.: Achievements and Challenges in Software Reverse Engineering. Communications of the ACM **54**(4), 142–151 (2011)
22. Chikofsky, E.J., II, J.H.C.: Reverse Engineering and Design Recovery: A Taxonomy. IEEE Software **7**(1), 13–17 (1990)
23. Chirila, C.B., Jebelean, C.: Towards Programs Logic Based Representation Driven by Grammar and Conforming to a Metamodel. In: Proceedings of the IEEE International Joint Conferences on Computational Cybernetics and Technical Informatics (ICCC-CONTI), pp. 107–112. IEEE Computer Society (2010)
24. Cho, E.S.: Integrated Meta-model Approach for Reengineering from Legacy into CBD. In: Proceedings of the International Conference on Computational Science and Its Applications (ICCSA), pp. 868–877. Springer-Verlag (2005)
25. Christopher, C.N.: Evaluating Static Analysis Frameworks. http://www.cs.cmu.edu/~aldrich/courses/654/tools/christopher-analysis-frameworks-06.pdf (2006)
26. Clark, T., Sammut, P., Willans, J.: Applied Metamodelling: A Foundation for Language Driven Development (Third Edition). ArXiv e-prints pp. 1–228 (2015)

27. Czarnecki, K., Helsen, S.: Classification of Model Transformation Approaches. In: Proceedings of the OOPSLA Workshop on Generative Techniques in the Context of Model-Driven Architecture, pp. 1–17 (2003). URL http://www.s23m.com/oopsla2003/mda-workshop.html
28. Czeranski, J., Eisenbarth, T., Kienle, H.M., Koschke, R., Plödereder, E., Simon, D., V, Y.Z., Girard, J., Würthner, M.: Data Exchange in Bauhaus. In: Proceedings of the 7th Working Conference on Reverse Engineering (WCRE), pp. 293–295. IEEE Computer Society (2000)
29. Demeyer, S., Ducasse, S., Tichelaar, S.: Why Unified is not Universal? UML Shortcomings for Coping with Round-trip Engineering. In: Proceedings of the 2nd International Conference on the Unified Modeling Language (UML), pp. 630–644. Springer-Verlag (1999)
30. Ducasse, S., Anquetil, N., Bhatti, U., Hora, A.C., Laval, J., Girba, T.: MSE and FAMIX 3.0: an Interexchange Format and Source Code Model Family. HAL 00646884 pp. 1–39 (2011)
31. Ducasse, S., Lanza, M., Tichelaar, S.: MOOSE: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems. In: Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools (COSET), pp. 1–7. IEEE (2000)
32. Durelli, R.S., Santibáñez, D.S.M., Delamaro, M.E., de Camargo, V.V.: Towards a Refactoring Catalogue for Knowledge Discovery Metamodel. In: Proceedings of the 15th IEEE International Conference on Information Reuse and Integration (IRI), pp. 569–576. IEEE Computer Society (2014)
33. Earley, J.: An Efficient Context-Free Parsing Algorithm. Communications of the ACM **13**(2), 94–102 (1970)
34. Ebert, J.: Metamodels Taken Seriously: The TGraph Approach. In: Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR), p. 2. IEEE Computer Society (2008)
35. Ebert, J., Kullbach, B., Riediger, V., Winter, A.: GUPRO - Generic Understanding of Programs, An Overview. Electronic Notes in Theoretical Computer Science **72**(2), 47–56 (2002)
36. Ebert, J., Kullbach, B., Winter, A.: GraX - An Interchange Format for Reengineering Tools. In: Proceedings of the 6th Working Conference on Reverse Engineering (WCRE), p. 89. IEEE Computer Society (1999)
37. Engström, E., Petersen, K.: Mapping Software Testing Practice with Software Testing Research - SERP-Test Taxonomy. In: Proceedings of the 8th IEEE International Conference on Software Testing, Verification and Validation (ICST) Workshops, pp. 1–4. IEEE Computer Society (2015)
38. Favre, J.M., Godfrey, M., Winter, A.: First International Workshop on Meta-Models and Schemas for Reverse Engineering ateM 2003. In: A. van Deursen, E. Stroulia, M.A.D. Storey (eds.) Proceedings of the 10th Working Conference on Reverse Engineering (WCRE), pp. 366–367. IEEE Computer Society (2003)
39. Favre, J.M., NGuyen, T.: Towards a Megamodel to Model Software Evolution Through Transformations. Proceedings of the Workshop on Software Evolution through Transformations: Model-based vs. Implementation-level Solutions (SETra), Electronic Notes in Theoretical Computer Science **127**(3), 59–74 (2005)
40. Favre, L.: Formalizing MDA-based Reverse Engineering Processes. In: Proceedings of the 6th International Conference on Software Engineering Research, Management and Applications (SERA), pp. 153–160. IEEE Computer Society (2008)
41. Ferenc, R., Beszédes, Á., Tarkiainen, M., Gyimóthy, T.: Columbus - Reverse Engineering Tool and Schema for C++. In: Proceedings of the 18th International Conference on Software Maintenance (ICSM), pp. 172–181. IEEE Computer Society (2002)
42. Ferenc, R., Sim, S.E., Holt, R.C., Koschke, R., Gyimothy, T.: Towards a Stardard Schema for C/C++. In: Proceedings of the 8th Working Conference on Reverse Engineering (WCRE), pp. 49–58. IEEE Computer Society (2001)
43. Garwick, J.V.: Programming Languages: GPL, A Truly General Purpose Language. Communications of the ACM **11**(9), 634–638 (1968)

44. Gîrba, T., Ducasse, S.: Modeling History to Analyze Software Evolution. Journal of Software Maintenance and Evolution: Research and Practice **18**(3), 207–236 (2006)
45. Glass, R.L.: Sorting Out Software Complexity. Communications of the ACM **45**(11), 19–21 (2002)
46. Gómez, V.U., Ducasse, S., D'Hondt, T.: Ring: A Unifying Meta-model and Infrastructure for Smalltalk Source Code Analysis Tools. Computer Languages, Systems and Structures **38**(1), 44–60 (2012)
47. Gómez, V.U., Kellens, A., Brichau, J., D'Hondt, T.: Time Warp, an Approach for Reasoning Over System Histories. In: Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (EVOL) Workshops, pp. 79–88. IEEE Computer Society (2009)
48. Gray, J., Zhang, J., Lin, Y., Roychoudhury, S., Wu, H., Sudarsan, R., Gokhale, A., Neema, S., Shi, F., Bapty, T.: Model-Driven Program Transformation of a Large Avionics Framework. Proceedings of the 3rd International Conference on Generative Programming and Component Engineering (GPCE), Lecture Notes in Computer Science **3286**, 361–378 (2004)
49. Guéhéneuc, Y., Albin-Amiot, H.: Using Design Patterns and Constraints to Automate the Detection and Correction of Inter-Class Design Defects. In: Proceedings of the 39th International Conference and Exhibition on Technology of Object-oriented Languages and Sytems (TOOLS), pp. 296–306. IEEE Computer Society (2001)
50. Harmer, T.J., Wilkie, F.G.: An Extensible Metrics Extraction Environment for Object-Oriented Programming Languages. In: Proceedings of the 2nd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM), pp. 26–35. IEEE Computer Society (2002)
51. Heidenreich, F., Johannes, J., Seifert, M., Wende, C.: Closing the Gap between Modelling and Java. Proceedings of the International Conference on Software Language Engineering (SLE), Lecture Notes in Computer Science **5969**, 374–383 (2010)
52. Holt, R.: An Introduction to TA: The Tuple Attribute Language. Department of Computer Science, University of Waterloo and Toronto pp. 1–10 (1998). URL http://plg.uwaterloo.ca/~holt/papers/ta.html
53. Holt, R.C., Schürr, A., Sim, S.E., Winter, A.: GXL: A Graph-based Standard Exchange Format for Reengineering. Science of Computer Programming **60**(2), 149–170 (2006)
54. Imber, M.: CASE Data Interchange Format Standards. Information and Software Technology **33**(9), 647–655 (1991)
55. Ishizue, R., Washizaki, H., Fukazawa, Y., Inoue, S., Hanai, Y., Kanazawa, M., Namba, K.: Metrics Visualization Technique Based on The Origins and Function Layers for OSS-based Development. In: Proceedings of the IEEE Working Conference on Software Visualization (VISSOFT), pp. 71–75. IEEE Computer Society (2016)
56. ISO/IEC: ISO/IEC 14977:1996 Information technology - Syntactic metalanguage - Extended BNF (1996)
57. ISO/IEC: ISO/IEC 25010:2011 Systems and Software Engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and Software Quality Models (2011)
58. ISO/IEC/IEEE: ISO/IEC/IEEE 42010:2011 Systems and Software Engineering - Architecture Description (2011)
59. Izquierdo, J.L.C., Molina, J.G.: A Domain Specific Language for Extracting Models in Software Modernization. In: Proceedings of the 5th European Conference on Model Driven Architecture-Foundations and Applications (ECMDA-FA), pp. 82–97. Springer-Verlag (2009)
60. Izquierdo, J.L.C., Molina, J.G.: An Architecture-Driven Modernization Tool for Calculating Metrics. IEEE Software **27**(4), 37–43 (2010)
61. Izquierdo, J.L.C., Molina, J.G.: Extracting Models from Source Code in Software Modernization. Software and Systems Modeling **13**(2), 713–734 (2014)
62. Jin, D.: Exchange Of Software Representations Among Reverse Engineering Tools. Technical Report 2001-454 pp. 1–131 (2001)
63. Jin, D., Cordy, J., Dean, T.: Where's The Schema? A Taxonomy of Patterns for Software Exchange. In: Proceedings of the 10th International Workshop on Program Comprehension (IWPC), pp. 65–74. IEEE Computer Society (2002)

64. Jin, D., Cordy, J.R.: Integrating Reverse Engineering Tools Using a Service-Sharing Methodology. In: Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC), pp. 94–99. IEEE Computer Society (2006)
65. Jouault, F., Bezivin, J.: KM3: a DSL for Metamodel Specification. Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems, Lecture Notes in Computer Science **4037**, 171–185 (2006)
66. Jouault, F., Bezivin, J., Kurtev, I.: TCS: a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering. In: Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE), pp. 249–254. ACM (2006)
67. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21 pp. 1–148 (1990)
68. Keller, R.K., Bédard, J., Saint-Denis, G.: Design and Implementation of a UML-Based Design Repository. In: Proceedings of the 13th International Conference on Advanced Information Systems Engineering (CAiSE), pp. 448–464. Springer-Verlag (2001)
69. Kienle, H.M., Müller, H.A.: Rigi - An Environment for Software Reverse Engineering, Exploration, Visualization, and Redocumentation. Science of Computer Programming **75**(4), 247–263 (2010)
70. Kitchenham, B., Brereton, O.P., Budgen, D., Turner, M., Bailey, J., Linkman, S.: Systematic Literature Reviews in Software Engineering - A Systematic Literature Review. Information and Software Technology **51**(1), 7–15 (2009)
71. Kitchenham, B.A., Dyba, T., Jorgensen, M.: Evidence-based Software Engineering. In: Proceedings of the 26th International Conference on Software Engineering (ICSE), pp. 273–281. IEEE Computer Society (2004)
72. Knodel, J., Calderon-Meza, G.: A Meta-Model for Fact Extraction from Delphi Source Code. Electronic Notes in Theoretical Computer Science **94**, 19–28 (2004)
73. Kollmann, R., Gogolla, M.: Capturing Dynamic Program Behaviour with UML Collaboration Diagrams. In: Proceedings of the 5th Conference on Software Maintenance and Reengineering (CSMR), pp. 58–67. IEEE Computer Society (2001)
74. Krasovec, G., Howell, S.: Applying the System Engineering Environment to the Reengineering Process. Journal of Systems Integration **5**(4), 309–336 (1995)
75. Kunert, A.: Semi-automatic Generation of Metamodels and Models From Grammars and Programs. Electronic Notes in Theoretical Computer Science **211**, 111–119 (2008)
76. Kurtev, I., Bézivin, J., Aksit, M.: Technological Spaces: An Initial Appraisal. In: International Symposium on Distributed Objects and Applications (DOA), pp. 1–6. Springer-Verlag (2002). URL http://doc.utwente.nl/55814/
77. Lanza, M.: CodeCrawler - Lessons Learned in Building a Software Visualization Tool. In: Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR), pp. 409–418. IEEE Computer Society (2003)
78. Lapierre, S., Laguë, B., Leduc, C.: Datrix$^{\text{TM}}$ Source Code Model and Its Interchange Format: Lessons Learned and Considerations for Future Work. ACM SIGSOFT Software Engineering Notes **26**(1), 53–56 (2001)
79. Lethbridge, T.C.: Requirements and Proposal for a Software Information Exchange Format (SIEF) Standard. http://www.site.uottawa.ca/˜tcl/papers/sief/standardProposal.html (1998)
80. Lethbridge, T.C., Tichelaar, S., Ploedereder, E.: The Dagstuhl Middle Metamodel: A Schema For Reverse Engineering. Electronic Notes in Theoretical Computer Science **94**, 7–18 (2004)
81. Lin, T., Robert Cheung, Z.H., Smith, K.: Exploration of Data from Modeling and Simulation through Visualization. In: Proceedings of the 3rd International SimTect Conference, pp. 303–308 (1998)
82. Lin, Y., Holt, R.C.: Formalizing Fact Extraction. Electronic Notes in Theoretical Computer Science **94**, 93–102 (2004)
83. Martinez, L., Pereira, C., Favre, L.: Recovering Sequence Diagrams from Object-oriented Code - An ADM Approach. In: Proceedings of the 9th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE), pp. 188–195. SciTePress (2014)

84. Meng, C., Wong, K.: A GXL Schema for Story Diagrams. Electronic Notes in Theoretical Computer Science **94**, 29–38 (2004)
85. Mens, T., Lanza, M.: A Graph-Based Metamodel for Object-Oriented Software Metrics. Electronic Notes in Theoretical Computer Science **72**(2), 57–68 (2002)
86. Minas, M.: Generating Visual Editors Based on Fujaba/MOFLON and DiaMeta. In: Proceedings of the 4th International Fujaba Days, pp. 35–42 (2006)
87. Naik, R., Bahulkar, A.: A Programmable Analysis and Transformation Framework for Reverse Engineering. Electronic Notes in Theoretical Computer Science **94**, 39–49 (2004)
88. Nierstrasz, O., Tichelaar, E., Demeyer, S.: CDIF as the Interchange Format between Reengineering. In: Proceedings of the OOPSLA Workshop on Model Engineering, Methods and Tools Integration with CDIF, pp. 1–8. ACM (1998)
89. OMG: Architecture-driven Modernization: Abstract Syntax Tree Metamodel (ASTM), Version 1.0 (2011)
90. OMG: Architecture-Driven Modernization: Knowledge Discovery Meta-Model (KDM), Version 1.3 (2011)
91. OMG: Meta Object Facility (MOF) Core Specification, Version 2.5 (2015)
92. OMG: XML Metadata Interchange (XMI), Version 2.5.1. http://www.omg.org/spec/XMI/2.5.1/ (2015)
93. Ossher, J., Bajracharya, S.K., Linstead, E., Baldi, P., Lopes, C.V.: SourcererDB: An Aggregated Repository of Statically Analyzed and Cross-linked Open Source Java Projects. In: Proceedings of the 6th International Working Conference on Mining Software Repositories (MSR), pp. 183–186. IEEE Computer Society (2009)
94. Owens, D., Anderson, M.: A Generic Framework for Automated Quality Assurance of Software Models - Application of an Abstract Syntax Tree. In: Proceedings of Science and Information Conference (SAI), pp. 207–211. IEEE (2013)
95. Pedro, L., Risoldi, M., Buchs, D., Barroca, B., Amaral, V.: Composing Visual Syntax for Domain Specific Languages. In: Proceedings of the 13th International Conference on Human-Computer (HCI), pp. 889–898. Springer-Verlag (2009)
96. Pérez-Castillo, R., de Guzmán, I.G.R., Gómez-Cornejo, R., Fernández-Ropero, M., Piattini, M.: ANDRIU. A Technique for Migrating Graphical User Interfaces to Android. In: Proceedings of the 25th International Conference on Software Engineering and Knowledge Engineering (SEKE), pp. 516–519. Knowledge Systems Institute Graduate School (2013)
97. Pinzger, M., Gall, H.C., Fischer, M.: Towards an Integrated View on Architecture and its Evolution. Electronic Notes in Theoretical Computer Science **127**(3), 183–196 (2005)
98. Reus, T., Geers, H., van Deursen, A.: Harvesting Software Systems for MDA-Based Reengineering. In: Proceedings of the Second European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA), pp. 213–225. Springer-Verlag (2006)
99. Saint-Denis, G., Schauer, R., Keller, R.K.: Selecting a Model Interchange Format: The SPOOL Case Study. In: Proceedings of the 33rd Annual Hawaii International Conference on System Sciences (HICSS), pp. 1–10. IEEE Computer Society (2000)
100. Santibáñez, D.S.M., Durelli, R.S., de Camargo, V.V.: A Combined Approach for Concern Identification in KDM Models. Journal of the Brazilian Computer Society **21**(1), 1–20 (2015)
101. Schauer, R., Keller, R.K., Lague, B., Knapen, G., Robitaille, S., Saint-Denis, G.: The SPOOL Design Repository: Architecture, Schema, and Mechanisms. In: H. Erdogmus, O. Tanir (eds.) Advances in Software Engineering, chap. 13, pp. 269–294. Springer-Verlag (2002)
102. Schürr, A.: Developing Graphical (Software Engineering) Tools with PROGRES. In: Proceedings of the 19th International Conference on Software Engineering (ICSE), pp. 618–619. ACM (1997)
103. Sharafi, Z., Soh, Z., Guéhéneuc, Y.: A Systematic Literature Review on the Usage of Eye-tracking in Software Engineering. Information and Software Technology **67**, 79–107 (2015)
104. Sim, S.E., Koschke, R.: WoSEF: Workshop on Standard Exchange Format. ACM SIGSOFT Software Engineering Notes **26**(1), 44–49 (2001)

105. Sim, S.E., Storey, M.A., Winter, A.: A Structured Demonstration of Five Program Comprehension Tools: Lessons Learnt. In: Proceedings of the 7th Working Conference on Reverse Engineering (WCRE), pp. 210–212. IEEE Computer Society (2000)
106. Smite, D., Wohlin, C., Galvina, Z., Prikladnicki, R.: An Empirically Based Terminology and Taxonomy for Global Software Engineering. Empirical Software Engineering **19**(1), 105–153 (2014)
107. Soden, M., Eichler, H.: An Approach to use Executable Models for Testing. In: Proceedings of the 2nd International Workshop on Enterprise Modelling and Information Systems Architectures (EMISA), pp. 75–85. GI (2007)
108. Sora, I.: A Meta-model for Representing Language-independent Primary Dependency Structures. In: Proceedings of the 7th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE), pp. 65–74. SciTePress (2012)
109. Sora, I.: Unified Modeling of Static Relationships between Program Elements. Proceedings of the 7th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE), Communications in Computer and Information Science **410**, 95–109 (2012)
110. Sora, I.: Helping Program Comprehension of Large Software Systems by Identifying Their Most Important Classes. Proceedings of the 10th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE), Communications in Computer and Information Science **599**, 122–140 (2015)
111. Sora, I., Todinca, D.: Using Fuzzy Rules for Identifying Key Classes in Software Systems. In: Proceedings of the IEEE 11th International Symposium on Applied Computational Intelligence and Informatics (SACI), pp. 317–322. IEEE (2016)
112. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E. (eds.): EMF: Eclipse Modeling Framework, 2nd Edition, 2 edn. Addison-Wesley Professional (2008)
113. Strein, D., Kratz, H., Löwe, W.: Cross-Language Program Analysis and Refactoring. In: Proceedings of the 6th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM), pp. 207–216. IEEE Computer Society (2006)
114. Tichelaar, S., Ducasse, S., Demeyer, S., Nierstrasz, O.: A Meta-model for Language-Independent Refactoring. In: Proceedings of the International Symposium on Principles of Software Evolution (ISPSE), pp. 154–164. IEEE Computer Society (2000)
115. Tilley, S.R., Wong, K., Storey, M.D., Müller, H.A.: Programmable Reverse Engineering. International Journal of Software Engineering and Knowledge Engineering **4**(4), 501–520 (1994)
116. Tripathi, V., Mahesh, T.S.G., Srivastava, A.: Performance and Language Compatibility in Software Pattern Detection. In: Proceedings of the IEEE International Advance Computing Conference (IACC), pp. 1639–1643. IEEE (2009)
117. Unterkalmsteiner, M., Feldt, R., Gorschek, T.: A Taxonomy for Requirements Engineering and Software Test Alignment. ACM Transactions on Software Engineering and Methodology **23**(2), 16:1–16:38 (2014)
118. Vidács, L.: Refactoring of C/C++ Preprocessor Constructs at the Model Level. In: Proceedings of the 4th International Conference on Software and Data Technologies (ICSOFT), pp. 232–237. INSTICC Press (2009)
119. W3C: Extensible Markup Language (XML). http://www.w3.org/XML/ (2000)
120. Washizaki, H., Fukazawa, Y.: A Technique for Automatic Component Extraction from Object-Oriented Programs by Refactoring. Science of Computer Programming **56**(1-2), 99–116 (2005)
121. Washizaki, H., Guéhéneuc, Y., Khomh, F.: A Taxonomy for Program Metamodels in Program Reverse Engineering. In: Proceedings of the 32nd IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 44–55. IEEE Computer Society (2016)
122. Washizaki, H., Guéhéneuc, Y., Khomh, F.: Patterns for Program Reverse Engineering from the Viewpoint of Metamodel. In: Proceedings of the 23rd Conference on Pattern Languages of Programs (PLoP), pp. 1–9. ACM (2016)
123. Wimmer, M., Kramler, G.: Bridging Grammarware and Modelware. Proceedings of the Satellite Events at the MoDELS Conference, Lecture Notes in Computer Science **3844**, 159–168 (2005)

124. Wu, H.: Test Case Generation for Programming Language Metamodels. Proceedings of the 1st Doctoral Symposium of the International Conference on Software Language Engineering (SLE), CEUR Workshop Proceedings **64**, 27–30 (2010)