A Taxonomy for Program Metamodels in Program Reverse Engineering

Hironori Washizaki
Dept. of Computer Science and Engineering
Waseda University, Tokyo, Japan
National Institute of Informatics, Tokyo, Japan
washizaki@waseda.jp

Yann-Gaël Guéhéneuc, Foutse Khomh
Ptidej Team, DGIGL Ecole Polytechnique de Montreal,
Quebec, Canada
{yann-gael.gueheneuc, foutse.khomh}@polymtl.ca

***Abstract*—Metamodels are frequently used during program reverse engineering activities to describe and analyze constituents and relations between the constituents of a program for supporting program comprehension, maintenance, and extension. Reverse engineering tools often define their own metamodels according to their own purposes and intended features. These metamodels have all advantages, and limitations that might have been solved by others. Although there are some existing works on the evaluation and comparison of these metamodels and tools, none of them consider all the possible characteristics and limitations to provide a comprehensive guidance for classification, comparison, reuse and extension of program metamodels. To guide practitioners and researchers to classify, compare, reuse, and extend program metamodels and their corresponding reverse engineering tools according to their goals, we first establish a conceptual framework with definitions of program metamodels and related concepts. Based on this framework, we provide a comprehensive taxonomy named Program Metamodel TAxonomy (ProMeTA), which incorporates characteristics that are newly identified into those that have already been stated in previous works identified by a systematic literature survey on program metamodels, while keeping the orthogonality of the entire taxonomy. We validate the taxonomy in terms of its orthogonality and usefulness through the classification of popular metamodels.**

***Index Terms*—reverse engineering, program metamodels, program comprehension and analysis, taxonomy**

I. Introduction

Many metamodels are used to describe and process software programs in program reverse engineering for program comprehension, maintenance, and extension. These metamodels are essential for the development of 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 own purposes and intended features [1]. Depending on the actual reverse engineering problem and the aspired program analysis technique, different code representations (i.e., metamodels) must be chosen. 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 and that the amount of extracted data by language analyzers (like compilers based on low-level metamodels) can become far too large to be comprehended or analyzed in a reasonable amount of time [2]. In contrast, for some analysis, details are essential for ensuring high precision and recall in the analysis results.

These metamodels have all advantages and limitations that might have been solved by others. By conducting a rigorous survey on program metamodels, we identified that these metamodels can be characterized by the following exhaustive orthogonal features: target language, abstraction level, metametamodel, exchange format, processing environment, definition, program meta and history data, and quality.

For example, regarding the abstraction level, there are low-level metamodels representing the complete code syntax, high-level ones representing abstract architectural elements, and mid-level ones representing neither the complete code syntax nor the architectural elements [3].

Because of differences between metamodels used, it is difficult to compare reverse engineering tools. These differences also lead to problems in exchanging information among the tools [4]. For example, it is known that fact extractors commonly do not agree and emit different facts on the same source program; this tends to undermine the users' understanding of the program and decrease their confidence in the extractor [4].

To guide practitioners and researchers to classify, compare, reuse and extend program metamodels and corresponding reverse engineering tools according to their goals, there exist some works on the evaluation and comparison of metamodels and tools such as [5], [6]. However, the comparisons and evaluations were conducted independently and do not give a comprehensive guidance of all the possible characteristics and limitations.

The goal of this paper is to provide a comprehensive taxonomy and to classify some popular metamodels according to this taxonomy. Our taxonomy, named Program Metamodel TAxonomy (ProMeTA), and classification results will 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 previously identified ones while keeping the orthogonality of the entire taxonomy.

We address the following research questions.

RQ1 Does ProMeTA cover all the possible characteristics and limitations in existing works on evaluation and comparison of program metamodels?

RQ2 Does ProMeTA have orthogonality of its classification features?

RQ3 Is ProMeTA useful for guiding practitioners and researchers? Possible usecases include making or choosing reverse engineering tools, and, communicating or researching on program metamodels and reverse engineering tools.

Our contributions are as follows:

- A conceptual framework for program reverse engineering from the viewpoint of metamodels.
- A comprehensive taxonomy (named ProMeTA) of features characterizing program metamodels in reverse engineering based on our framework.
- A classification of existing popular program metamodels based on our taxonomy.

The remainder of this paper is organized as follows. We provide some background in Section II. In Section III, we propose our conceptual framework. In Section IV, we show our taxonomy, which we validate and discuss in Section V. Finally, we conclude our work and discuss future work in Section VI.

II. Background

A. Program reverse engineering

Reverse engineering is the process of analyzing a subject system to identify the system's constituents and creating representations in another forms or at higher levels of abstraction [7]. Although reverse engineering can be started from any level of abstraction, this paper focuses on *program reverse engineering*, i.e., the process of analyzing the program source code to identify the program's constituents and create a representation of the program. It is motivated by the fact that, when maintaining a software system, the only reliable information is often embedded in the source code of the program [8].

Moreover, this paper limits the target program codes to those written in general purpose programming languages (GPLs) [9] such as C and Java. GPLs are used to solve a broad spectrum of problems [10] in comparison to domain specific languages (DSLs), which are used for particular problems. DSLs usually offer higher-level constructs (e.g., rules) in comparison to GPLs [11]. Thus, it is challenging to have appropriate metamodels for describing GPL programs in appropriate abstraction levels according to specific purposes such as program analysis, visualization, etc.

B. Program Metamodel

Fact extraction from source codes is aimed at finding pieces of information about the system (e.g., the name of a class or what function calls what function) [12]. 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 [12]. Essential to a fact extractor is the underlying *metamodel* (i.e., schema), which specifies the constituents and relations to be extracted [12].

In addition, schemas are essential for the development of reverse engineering tools since they also specify the underlying semantic model of various analysis services [13]. Here, from the viewpoint of modeling technology, schemas for fact extraction from program are regarded as *program metamodels* while the extracted facts are regarded as models of the programs that conform to the corresponding schemas used for extraction.

III. Terminology and Conceptual Framework

Program metamodels are used under various contexts (such as forward engineering and reverse engineering) and at various abstraction levels (from architecture to code). Yet, the concept of metamodels is often not clearly recognized. Indeed, there are many synonyms for "metamodel" including "schema", "representation", "format", and "form". Moreover, metamodels are often discussed together with standard exchange formats (SEFs) without a clear distinction between these two concepts. For example, Sim and Koschke [2] report that the workshop focused on SEFs had a presentation addressing "a family of related SEFs including MOF, XMI, UML, XML and CDIF." However, Meta Object Facility (MOF) [14] is a metametamodel while the others are basically SEFs, although UML can also serve as a program metamodel.

To establish a common vocabulary, we first define the core concepts below.

- A *model* is a simplification of a system with an intended goal in mind [15]. For example, a diagram showing only the program modules and their dependencies is a model of a program that has been created with the goal of understanding the basic structure of the program at a higher level of abstraction.
- A *metamodel* is a model of a language that captures the essential properties and features [16] of some target models. Although metamodels have primarily been developed and advertised by the Object Management Group (OMG) with its MOF standard [17] in the context of modelware, metamodels are not limited to MOF-based models. *Program metamodels* in modelware [18], *schemas (or exchange format)* in dataware, and *grammars* in grammarware [19] are all metamodels [15] in different technological spaces [18], [20]; these are models of program modeling languages, data languages, and programming languages, respectively.

According to the above-mentioned concepts, program metamodels and related concepts are defined as follows, and Figure 1 shows the relationships among them following the OMG four-layer metamodel hierarchy [18], [14] 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; a program model must conform

to its program metamodel. Examples include Knowledge Discovery Meta-Model (KDM)[21], FAMOOS Information Exchange Model (FAMIX) [22] and UML.

- A program *meta-language* is a language to describe program metamodels. Meta-languages can be classified into two types: *metasyntaxes of grammar* (such as Extended BNF (EBNF) [23]), which are textual, and *meta-metamodels* of metamodels at certain abstraction levels (such as MOF and Eclipse Modeling Framework (EMF) meta model Ecore [24]), which are graphic.
- A *context-free grammar* (or simply *grammar*) is a formal device for specifying which strings are part of the language, where a language is a set of strings over a finite set of symbols [25].
- 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 [26].
- An *abstract syntax tree (AST)* is a simplified syntactic representation of a source code which excludes superficial distinctions of form and constituents that are unimportant for translation from the tree [26]. 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 graphic representation of an abstract syntax (tree). Abstract syntax models can be seen as low-level program metamodels. Examples of abstract syntax models are programming-language-independent AST models such as ASTM [27] and programming-language-specific AST models such as Java Metamodel [28].
- A *standard exchange format (SEF)* (or simply *exchange format*) is a metamodel (i.e., schema) of model data used to store the data that are exchangeable among different tools. Examples include XML, XML Metadata Interchange format (XMI), Resource Descriptor Format (RDF), Rigi Standard Form (RSF), Tuple-Attribute Language (TA), GraX [2], and CASE Data Interchange Format (CDIF) [29]. Some of these (such as XMI and RDF) are general-purpose exchange formats that can be adapted to software, while others are specifically for software [2].

IV. Taxonomy Construction

Based on the background described in Section II and the vocabulary presented in Section III, we identified various characteristics to distinguish existing program metamodels. We propose a comprehensive taxonomy, named Program Metamodel TAxonomy (ProMeTA) for the classification of program metamodels in the form of feature diagrams based on our conceptual framework. ProMeTA integrates characteristics stated in existing works with those that we have newly identified while keeping the orthogonality of the entire taxonomy.

We describe ProMeTA and its construction process in detail below.

A. Taxonomy Construction Process

The development of a taxonomy can be approached in two different ways: top-down or bottom-up [30], [31]. In the top-down approach, a taxonomy is built upon existing knowledge structures, allowing the reuse of established definitions and categorizations and hence increasing the probability of achieving an objective classification procedure [30].

As we mentioned, there are some existing works on the evaluation and comparison of program metamodels and tools but none of them gives us a comprehensive guidance that takes into account all the possible characteristics and limitations. Therefore, we basically adopt the top-down approach to design ProMeTA based on our conceptual framework as follows.

1) A specific taxonomy is designed to accommodate a single, well-defined purpose while it is applicable in various circumstances [30]. Therefore, we start by clearly defining the specific purpose of ProMeTA – to support stakeholders to classify, compare, reuse and extend program metamodels in program reverse engineering. Additionally, the taxonomy is expected to support communication among all stakeholders, thereby increasing the accessibility of research results in program metamodels and reverse engineering.
2) For the top-down approach, we identified existing works on classification and quality properties of program metamodels and tools by a systematic literature review (SLR). During the SLR, we also identified several popular metamodels. The aim of an SLR is to aggregate all existing evidence on a research question and support the development of evidence-based guidelines for researchers and practitioners [32].
   Then, we analyzed existing classifications and comparisons on program metamodels and related concepts [33], [34], [35], [36], [37], [38], [3], [39], [40], [5], [6] that are identified by the SLR, and merged them together into one structure in the form of feature diagrams [41] by referring to the basic term classification defined in our conceptual framework. We also merged quality properties of program metamodels and related concepts discussed in 12 papers [42], [36], [43], [44], [38], [45], [18], [46], [13], [47], [48], [16] that are identified by the SLR. A feature diagram essentially defines a taxonomy [46].
3) In addition to the above-mentioned existing knowledge structures, we added all the characteristics identified in existing metamodels to the taxonomy while keeping its orthogonality by referring to the basic term classification defined in the framework.
4) Finally, we validated the taxonomy in terms of its orthogonality, coverage and usefulness by using it to classify the five popular metamodels identified in the SLR.

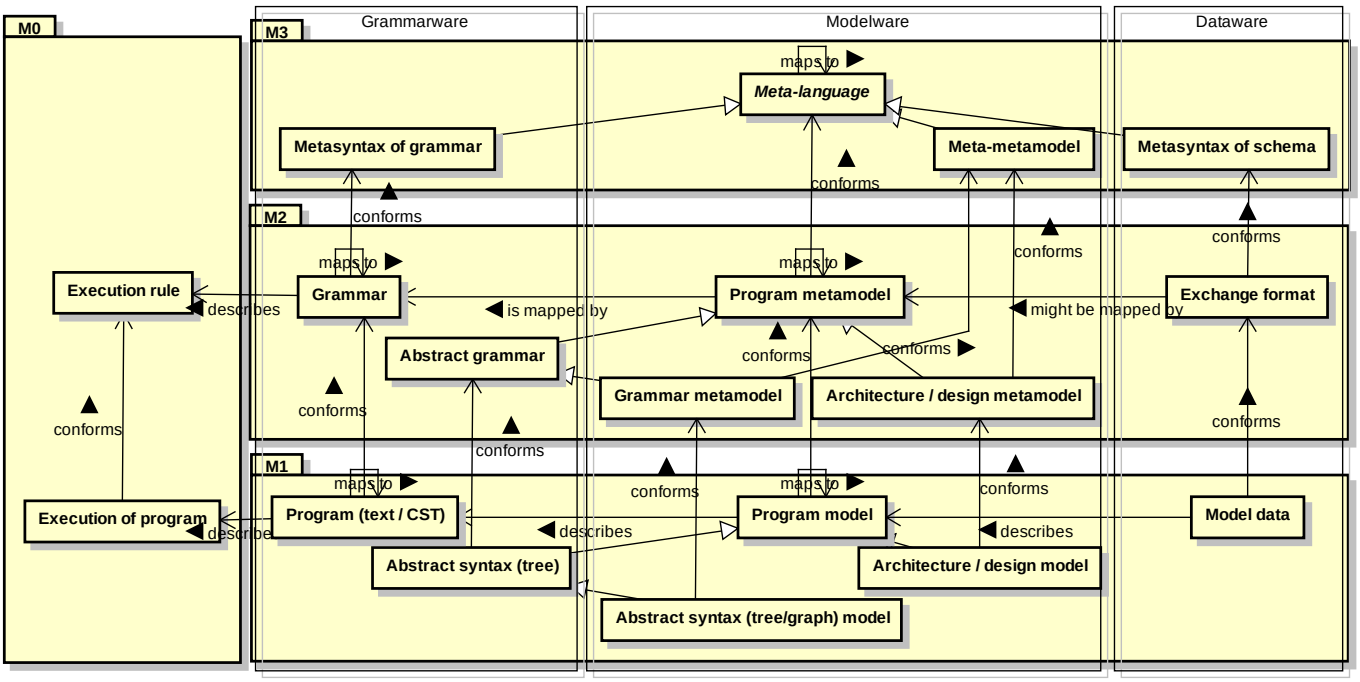

Fig. 1. Conceptual framework of program metamodels

B. Systematic literature review

We search for papers that are about program metamodels in reverse engineering by using Engineering Village[1]. Engineering Village is a search platform providing access to 12 trusted engineering document databases such as Ei Compendex and Inspec.

We used our search query as shown in Figure 2; it is simple but expected to be enough to find relevant papers since any reverse engineering objective and application have to employ some sort of transformations; extraction and generation can be regarded as a kind of vertical transformations [49].

We initially obtained 1234 papers by executing our query in Engineering Village as of October 13th 2015. After that, we adopt the following inclusion and exclusion criteria. We go through the title and abstract (and the body if necessary) to check the relevance of selection.

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 include studies on program reengineeing such as modernization and refactoring only if they employ program metamodels for explicit reverse engineering phase as a part of entire reengineering process.

b). Studies that present details and/or complete results if there are more than one studies on the same topic reported by the same author group.

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

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

Fig. 2. The search query

Exclusion criteria:

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

b). Studies that are not directly related to program reverse engineering targeting program source code written in GPLs.

c). Elements of ”grey” literature that are not published by trusted, well-known publishers, and did not use a well-defined referee process[50].

d). Articles that are not published in English.

Moreover, by performing the snowballing process, we finally obtained a set of 62 papers. Among them, 11 papers are about classification and comparison on program metamodels and related concepts [33], [34], [35], [36], [37], [38], [3], [39], [40], [5], [6], while 12 papers are about quality properties [42], [36], [43], [44], [38], [45], [18], [46], [13], [47], [48], [16].

C. Overview of Taxonomy

ProMeTA consists of nine features that represent major points of variation as shown in Figure 3. We detail each feature below.

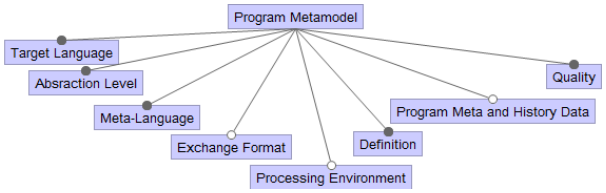

Fig. 3. Feature diagram for program metamodels

D. Feature: Target Language

Language independence varies from metamodel to metamodel; some metamodels only handle certain languages while others handle multiple languages in a certain or any category. Moreover, even if a metamodel is stated to be "language independent", we often found that it actually only supports a very limited number of languages at the time of our analysis. To define these characteristics precisely, the target language feature consists of two parts as shown in Figure 4: language independence and current supported languages.

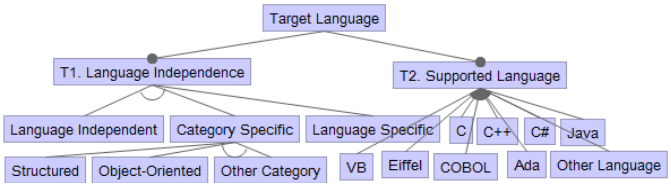

Fig. 4. Feature diagram for target languages

E. Feature: Abstraction Level

A representation (i.e., model) conforming to a metamodel must be as abstract as possible [51] within the limits of its reverse engineering objectives. Metamodels can be classified into three abstraction levels as shown in Figure 5: 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 the metamodel represents neither of the above [3].

According to the requirements for SEFs [35], SEFs should be able to deal with classes (i.e., modules), associations (i.e., relationships) and attributes. The same requirements can be commonly applied to high- or mid-level program metamodels as well; the domain ontology for integrating several reverse engineering tools (based on high- or mid-level metamodels) [5] 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 [5], that are applicable to the mid-level metamodels.

Regarding the low-level metamodels, we follow the three representation aspects [37]: Lexical Structure, Syntax and Semantics. Moreover, we added Dialects such as non-standard language specifiers as well as Preprocessor Artifacts [38] and Static/Dynamic semantics [40], taken from existing schema comparisons [38], [40].

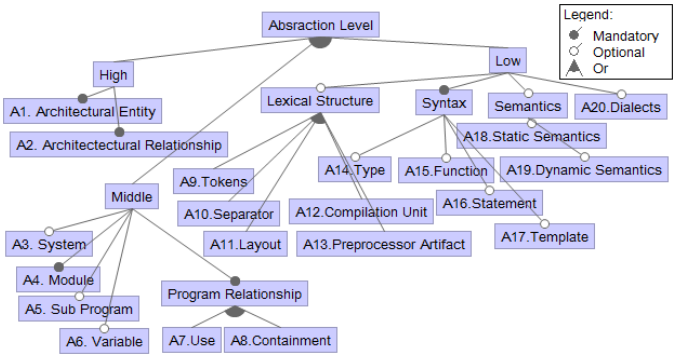

Fig. 5. Feature diagram for abstraction levels

F. Feature: Meta-Language

The data structures of SEFs used to represent software are classified into three types [44]: Tree, Graph and Structured Data (i.e., data that is not a tree or a graph). We adopt the same classification for classifying meta-languages aligned with our conceptual framework.

Based on the classification shown in Figure 6, we list several well-accepted standard meta-metamodels together with the metasyntax of grammar, including MOF, EMF/Ecore, Kernel MetaMetaModel (KM3) [52], UML and EBNF. KM3 is a meta-metamodel that has concepts similar to those found in MOF but is simpler than MOF [11]. Although UML is originally a modeling language classified in the M2 layer of the OMG four-layer metamodel hierarchy, it is often used for modeling program metamodels.

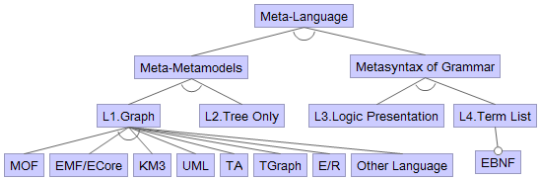

Fig. 6. Feature diagram for meta-languages

G. Feature: Exchange Format

Program metamodels could depend on or have a high affinity with specific SEFs, although it is preferable for them to be independent from any SEF in order to exchange models among tools. For example, a reverse engineering tool environment called MOOSE [53] defines its own program metamodel called FAMIX, but it adopts CDIF (and later XMI) to exchange FAMIX-based information between different tools [54], [44].

Figure 7 shows the characteristic properties and considerations of SEFs [44], [45]. Among them, most of the quality characteristics, including scalability, simplicity, neutrality, formality, flexibility, evolvability, identity, solution reuse and legibility, are basically examined according to the exchange patterns [45], i.e., combinations of the clarity and locality of the exchange format on which the metamodel depends.

The exchange format satisfies the integrity only if some special mechanism for ensuring errorless exchange has been provided with the exchange format on which the metamodel depends [45]. The exchange format satisfies the popularity if many different tools support the format. The exchange format satisfies the completeness only if all the information in the metamodel can be included in the exchange format. The exchange format satisfies the transparency only if no loss, alteration or gain in the information transferred occurs due to the use of encoders and decoders of the exchange format [45].

As for the property of the Abstract Syntax property, we list well-accepted SEFs, including Annotated Terms (ATerms) [55], [56], InterMediate Language (IML) and Resource Graph (RG) [57], Multi-Layer and Multi-Edge-Set (MLMES) graph [58], CASE Data Interchange Format (CDIF) [29], Tuple-Attribute Language (TA) [59], TA++ [35] and Datrix-TA [60], PROgramming with Graph Rewriting Systems (PROGRES) graph specification [61], GraX/TGraph [62], Graph Exchange Language (GXL) [63], and Rigi Standard Form (RSF) [64], along with general-purpose exchange formats, including XML [65] and XMI [66].

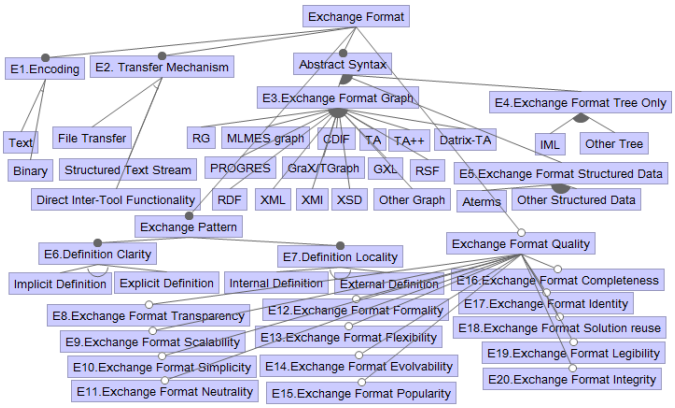

Fig. 7. Feature diagram for exchange formats

H. Feature: Processing Environment

By providing mechanisms to query (i.e., navigate) and transform program models, language toolkits including reverse engineering tools can fulfil analysis and comprehension tasks, as well as carry out maintenance and source code transformation tasks [67]. Specific processing environments to provide such mechanisms for navigation, transformation, analysis and extraction are often provided together with the program metamodels. Figure 8 represents the major points of variation for processing environment.

I. Feature: Definition

Program metamodels are mostly defined manually. Some approaches exist for generating program metamodels automatically from grammars [51], [68], although these are originally intended to work for DSLs. Regarding the clarity and locality of the definition, program metamodels can be classified into

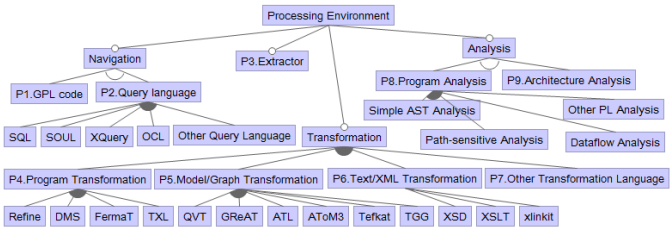

Fig. 8. Feature diagram for processing environments

four exchange patterns, similar to SEFs [44], [45]: implicitly-internally defined, implicitly-externally defined, explicitly-internally defined, and explicitly-externally defined. Figure 9 shows these characteristic properties.

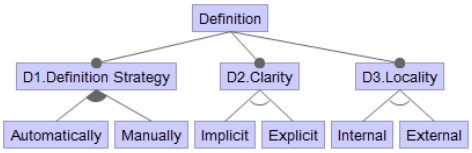

Fig. 9. Feature diagram for definition

J. Feature: Program Meta and History Data

According to the requirements for SEFs [35], SEFs should be able to store basic data (i.e., meta data) about the software systems they represent, including programming language versions, software system versions, dates of files creation and versions of files. We believe that program metamodels are also expected to handle such meta data together with the name of the programming languages.

Moreover, several program metamodels, such as Ring [69], directly support the history data, which allow reverse engineering tools to work easily with source code versioning systems to conduct history analysis at some abstraction levels. Figure 10 shows these characteristic properties.

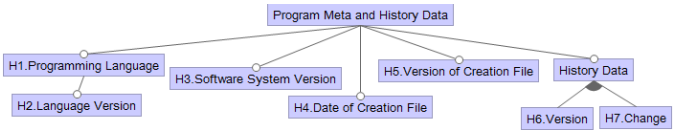

Fig. 10. Feature diagram for program meta and history data

K. Feature: Quality

We use the standard quality model ISO/IEC 25010:2011 [70] as the basis for specifying quality properties of program metamodels in a comprehensive and consistent manner. During the SLR, we found 12 papers discussing quality properties that are applicable to program metamodels, which are: requirements for SEFs [43], [44], [45], requirements for C++ schemas [38], requirements for reverse engineering tools enabled by schemas[42], [13], comparative considerations for program

comprehension tools [36], evaluation properties for static analysis frameworks [47], comparative issues for technological spaces [18], tracing features for model transformations [46], formality levels of metamodeling [16], and correctness of metamodels [48].

We categorize these properties together with those that we have 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 11 shows the feature diagram for functional suitability, while Figure 12 shows the feature diagram for the other quality characteristics. We summarize them as follows:

- Functional suitability consists of three sub-characteristics: 1) functional appropriateness, which is mostly concerned with traceability [18], [46] from model elements to the corresponding portion of the source code, 2) functional correctness regarding how the program metamodel has been verified [48], and 3) functional completeness regarding the applicability of the metamodel (i.e., general purpose metamodels or task-specific ones) [42]. In general, low-level metamodels are good for executability since any GPL should provide executable semantics, while most mid- or high-level metamodels lack executable semantics.
- Performance efficiency addresses the quantity of extracted data [36], which primarily depends on the granularity of the metamodel. A metamodel sacrifices such resource utilization if the ratio of 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 [45]. A metamodel satisfies the integrity only if some special mechanism for ensuring errorless exchange has been provided with the metamodel [45]. A metamodel satisfies the instance representation [38] if the metamodel instance (i.e., a model) can be easily represented in any SEF; this property is almost identical to the content-presentation separation [18].
- Usability addresses the learnability that is supported by the existence of documentations, samples and user communities [47].
- Reliability addresses the availability of the program metamodel in terms of licensing [47]. Metamodels are expected to be fully available through websites or other means; sometimes, however, only parts of a metamodel are provided.
- Maintainability encompasses five sub-characteristics. Among them, simplicity and evolvability are primarily determined by the exchange patterns [45]. Some metamodels have specific modularity mechanisms (such as packages) and/or reuse mechanisms (such as the inheritance and logical composition of metamodel elements) [46] to improve maintainability. The formality is specified as partially formalized or completely formalized [16] according to the available metamodel definition.
- Portability addresses the adaptability and is composed of three concrete properties. Among them, flexibility and scalability are primarily determined by the exchange patterns [45]. A metamodel satisfies the popularity if many different organizations other than the original developers have used it.

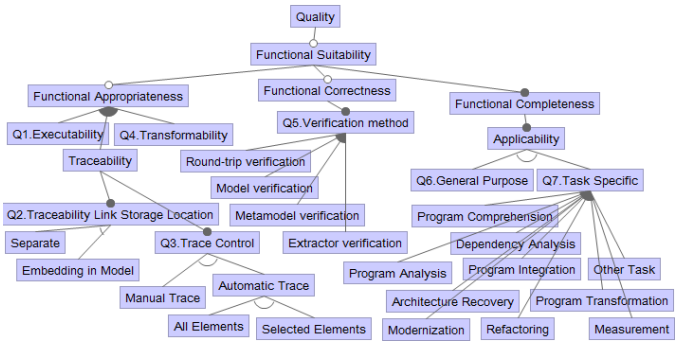

Fig. 11. Feature diagram for functional suitability

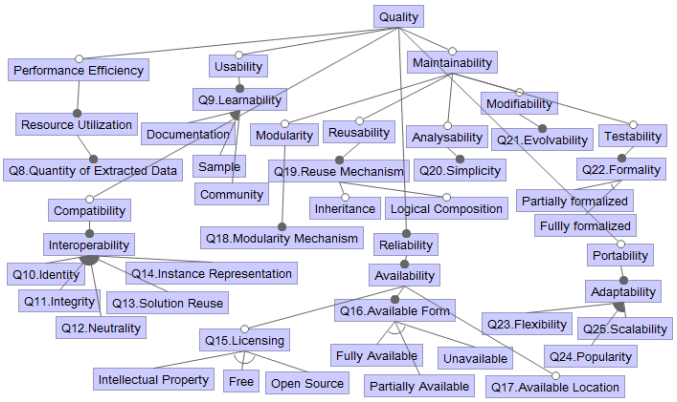

Fig. 12. Feature diagram for performance efficiency, compatibility, usability, reliability, maintainability and portability

V. Validation of ProMeTA

A taxonomy can be validated by demonstrating the orthogonality of its classification features, by benchmarking against existing classification schemes, and by demonstrating its utility to classify existing knowledge [71]. We validated ProMeTA by classifying popular metamodels identified in the SLR.

A. Target popular metamodels

In the set of 62 papers obatained by the SLR, the following five program metamodels are adopted to concrete reverse engineering techniques or tools in multiple papers:

- M1. Abstract Syntax Tree Metamodel (ASTM): two papers [6], [72]
- M2. Knowledge Discovery Meta-Model (KDM): four papers [73], [74], [75], [76]

- M3. FAMOOS Information Exchange Model (FAMIX): five papers [77], [78], [79], [80], [81]
- M4. SPOOL Metamodel: two papers [82], [83]
- M5. UNIQ-ART Metamodel: two papers [84], [85]

B. Classification result

We classified the above-mentioned five metamodels M1–M5 by using ProMeTA. Figure 1 shows the classification result. We summarize the findings and corresponding suggestions for practitioners and researchers as the followings.

- Target language: Three of the five metamodels are language independent while two are specific to handle object-oriented program code. And regardless of the language independence, all of them support Java language since it is thought to be the most common language especially in the context of reverse engineering research and practice. And the second common language is C++. If the target language is major like Java and C++, practitioners and researchers could reuse many of existing program metamodels and their corresponding reverse engineering tools while they have to choose specific metamodel or create new one if the target language is minor.
- Abstraction level: All of the five metamodels can be used as the mid-level metmodels while only single metamodel (M2) can be used as the high-level one. According to the coverage of the low-level metamodel features, M1 and M2 are more useful as the low-level metamodels although they still miss some lexical structure features such as Token, Separator and Layout. There is no metamodel among five that directly supports and implements language dialects.
  Practitioners and researchers could choose an appropriate metamodel and its corresponding reverse engineering tool according to their requirements on the abstraction level. However, there could no existing metamodel supporting all of required features at certain abstraction levels, especially at the low-level according to our classification results; in that case, it could be necessary to extend existing metamodels or create new one to cover those missing features.
- Meta-language: Four of the five metamodels adopt the standard meta-metamodel MOF or the language UML that are explicitly and externally defined, while only M5 adopts a specific implicit and internal one.
  If practitioners and researchers consider to adopt various tools for long-term usage, it could be better to choose or create program metamodels (like M1–M4) defined by the widely-accepted, explicit and external meta-languages such as MOF and UML.
- Exchange format: As corresponding to the meta-language used, four of the five metamodels adopt the standard SEFs such as XMI that are explicitly and externally defined, while only M5 supports a specific binary-based implicit and internal data exchange.
  If practitioners and researchers consider to utilize various tools for long-term usage, it could be better to choose or create program metamodels with good exchange format quality (like M1–M4), that support the widely-accepted, explicit and external SEFs such as XMI.
- Processing environment: There are dedicated extractors and navigation supports for all of the five metamodels since these are popular ones; extractors and navigation supports should be prepared to improve the ease of use of metamodels. There are dedicated transformation supports including refactoring facilities for three of five. Most of the metamodels except for M5 are suitable for program analysis.
  Practitioners and researchers should check whether processing environment and facilities are available to meet their reverse engineering objectives.
- Definition: All of the five metamodels are manually defined. Most of them except for M5 are explicitly defined; it leads to high quality of the metamodels, especially high compatibility, maintainability and portability. Three of five are externally and fully formalized while two (M4 and M5) are internally defined.
  If practitioners and researchers consider to utilize various tools for long-term usage, it could be better to choose or create program metamodels that are explicitly and externally defined (like M1–M3).
- Program meta and history data: There are very little support for describing meta and history data in metamodels; only the programming language name and the file version are supported by M1 and M2 respectively.
  During the SLR, we found several history-aware metamodels that explicitly address version history: Ring [69], Hismo [86], [87], FAMIX-based RHDB code model [88] and FAMIX-based ArchEvoDB schema [89]. If practitioners and researchers have to conduct reverse engineering such that history analysis is taken into account, it could be better to choose these history-aware metamodels; especially RHDB code model [88] and ArchEvoDB schema [89] could be recommended since these are extension of FAMIX, which is the widely-accepted popular metamodel.
- Functionality: Two (M1 and M2) of the five metamodels support most of 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 of five explicitly state how these have been verified. Most of five can be used for various purposes while only M5 is specific to the dependency analysis purpose.
  Practitioners and researchers should check whether program metamodels under consideration satisfy their reverse engineering functionality requirements. And if they have to use metamodels for various reverse engineering purposes, it is better to choose general purpose metamodels (like M1–M4).
- Non-functionality: Only M1 sacrifices the performance

efficiency since it contains all of statement-level code descriptions. Three (M1–M3) of the five metamodels have good usability since documents, 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 definitions of M4 and M5 seem to be unavailable elsewhere on web nor literatures. Most of the metamodels except for M5 support the inheritance and logical composition as the reuse mechanism. Only M2 supports the dedicated modularity mechanism.

Practitioners and researchers should check whether program metamodels under consideration satisfy their non-functionality requirements. And it is obvious that they have to choose fully available and formalized metamodels if they want to reuse existing ones.

C. Discussion

RQ1: Does ProMeTA cover all the possible characteristics and limitations in existing works on evaluation and comparison of program metamodels?

During the construction process of ProMeTA, all of the important characteristics from existing classification schemes/frameworks and comparison [33], [34], [35], [36], [37], [38], [3], [39], [40], [5], [6] and discussions on quality properties [42], [36], [43], [44], [38], [45], [18], [46], [13], [47], [48], [16] for program metamodels and related concepts, which are identified by the SLR, were included or mapped to the items in ProMeTA, implying its adequate coverage by construction. Thus, it is implicitly benchmarked against existing classification schemes.

RQ2: Does ProMeTA have orthogonality of its classification features?

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

RQ3: Is ProMeTA useful for guiding practitioners and researchers?

We now discuss how ProMeTA could guide practitioners and researchers in the following possible usecases UC1–UC3 below.

- UC1. Making new reverse engineering tools: When engineers want to build their own new reverse engineering tools, they first have to define their own requirements on program metamodels that enable and circumscribe the features of the tools. ProMeTA supports the requirements definition and guides further reuse, extension or creation of metamodels since engineers can recognize features included in ProMeTA as possible requirement items. Moreover, if a ProMeTA-based classification result of a metamodel under consideration of reuse or extension is available like M1–M5 in the above validation, engineers can easily decide whether the metamodel satisfies their own requirements based on ProMeTA.
- UC2. Choosing existing reverse engineering tools: When engineers want to reuse and eventually extend existing reverse engineering tools, they have to compare and choose appropriate one according to how the underlying program metamodels meet their objectives. ProMeTA could guide such comparison by providing them important comparison criteria and existing classification results of metamodels if available.
- UC3. Communicating or researching on program metamodels and reverse engineering tools: ProMeTA can serve as a reference for the reverse engineering community including practitioners and researchers, and can be easily extended by peers, which in result will provide the community with an important body of knowledge to guide future communications and researches on program metamodels and corresponding reverse engineering tools since it incorporates characteristics of metamodels into the single orthogonal structure based on the conceptual framework that defines the common terminology. For example, ProMeTA could be a 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 could be a starting point of building that.

D. Limitation

The classification of the five popular metamodels based on ProMeTA was conducted by the first author of this paper and quickly reviewed by the second and third authors. Therefore, there is a possibility that our classification results may not be completely correct; to mitigate this threat to validity, we have a plan to open the classification results together with ProMeTA to the public and call for comments on them. Especially, we need to contact with original developers of the metamodels addressed in the paper and request review.

Although we identified characteristics of program metamodels during the rigorous systematic literature survey, we might miss some other characteristics to be used for classification of metamodels. ProMeTA is expected to efficiently incorporate such missing or future characteristics newly identified to the single structure since we believe that the form of feature diagrams could make such extension of the taxonomy easy.

Any taxonomy can only unleash its full potential through widespread awareness and a large number of contributions [90]. Therefore, our future work is to follow a popularization strategy [90].

VI. Conclusion and Future Work

In this paper, we proposed a conceptual framework with definitions of program metamodels and related concepts, and built a comprehensive taxonomy named ProMeTA based on this framework. ProMeTA incorporates the characteristics that we have newly identified into those that have been stated in

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	
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	XMI, CDIF			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, KDM Analysis Package	MoDisco (dedicated parsers), Gra2MoL	KDM Target Mapping & Transformation Package				X	
M2		OCL, Modisco Java Model Query	MoDisco (KDM Source Discovery, Java Discoverer)		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 Meta 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														Dependency analysis

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. 13. Classification result by using ProMeTA (X: supports the characteristic indicated, ++: particularly satisfies the characteristic/requirement indicated, +: satisfies the characteristic/requirement indicated, -: sacrifices or does not satisfies the characteristic/requirement indicated, Exp: Explicit, Imp: Implicit, Ext: External, Int: Internal)

existing works identified by the systematic literature survey on program metamodels, while keeping the orthogonality of the entire taxonomy, which is accomplished by referring to the basic term classification defined in the framework. We validated the taxonomy in terms of its orthogonality and usefulness through the classification of popular metamodels from the survey.

As our future work, we will make ProMeTA available to the reverse engineering community, including practitioners and researchers, through scientific publications and our Website[2]. We can then achieve a widely agreed-upon view of the terminology and classification characteristics used in the taxonomy, and receive further input to extend the taxonomy with new categories and data sets to reflect its usage.

REFERENCES

[1] J. Ebert, B. Kullbach, V. Riediger, and A. Winter, "Gupro - generic understanding of programs, an overview," *Electronic Notes in Theoretical Computer Science*, vol. 72, no. 2, pp. 47–56, 2002.

[2] S. E. Sim and R. Koschke, "Wosef: Workshop on standard exchange format," *ACM SIGSOFT Software Engineering Notes*, vol. 26, no. 1, pp. 44–49, 2001.

[3] T. C. Lethbridge, S. Tichelaar, and E. Ploedereder, "The dagstuhl middle metamodel: A schema for reverse engineering," *Electronic Notes in Theoretical Computer Science*, vol. 94, pp. 7–18, 2004.

[4] Y. Lin and R. C. Holt, "Formalizing fact extraction," *Electronic Notes in Theoretical Computer Science*, vol. 94, pp. 93–102, 2004.

[5] D. Jin and J. R. Cordy, "Integrating reverse engineering tools using a service-sharing methodology," in *14th IEEE International Conference on Program Comprehension (ICPC'06)*. IEEE Computer Society, 2006, pp. 94–99.

[6] J. L. C. Izquierdo and J. G. Molina, "Extracting models from source code in software modernization," *Software and Systems Modeling*, vol. 13, no. 2, pp. 713–734, 2014.

[7] E. J. Chikofsky and J. H. C. II, "Reverse engineering and design recovery: A taxonomy," *IEEE Software*, vol. 7, no. 1, pp. 13–17, 1990.

[8] G. Canfora, M. D. Penta, and L. Cerulo, "Achievements and challenges in software reverse engineering," *Communications of the ACM*, vol. 54, no. 4, pp. 142–151, 2011.

[9] J. V. Garwick, "Programming languages: Gpl, a truly general purpose language," *Communications of the ACM*, vol. 11, no. 9, pp. 634–638, 1968.

[2]http://www.washi.cs.waseda.ac.jp/prometa/

[10] T. Buchner and F. Matthes, "Introspective model-driven development," *Third European Workshop on Software Architecture (EWSA 2006), Revised Selected Papers, Lecture Notes in Computer Science*, vol. 4344, pp. 33–49, 2006.

[11] F. Jouault, J. Bezivin, and I. Kurtev, "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'06)*. ACM, 2006, pp. 249–254.

[12] J. Knodel and G. Calderon-Meza, "A meta-model for fact extraction from delphi source code," *Electronic Notes in Theoretical Computer Science*, vol. 94, pp. 19–28, 2004.

[13] J.-M. Favre, M. Godfrey, and A. Winter, "First international workshop on meta-models and schemas for reverse engineering atem 2003," in *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE'03)*, A. van Deursen, E. Stroulia, and M.-A. D. Storey, Eds. IEEE Computer Society, 2003, pp. 366–367.

[14] OMG, "Omg meta object facility (mof) core specification, version 2.5," 2015.

[15] J.-M. Favre and T. NGuyen, "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 2004), Electronic Notes in Theoretical Computer Science*, vol. 127, no. 3, pp. 59–74, 2005.

[16] T. Clark, P. Sammut, and J. Willans, "Applied Metamodelling: A Foundation for Language Driven Development (Third Edition)," *ArXiv e-prints*, 2015.

[17] M. Alanen and I. Porres, "A relation between context-free grammars and meta object facility metamodels," *TUCS Technical Report, No.606*, 2003.

[18] I. Kurtev, J. Bézivin, and M. Aksit, "Technological spaces: An initial appraisal," in *International Symposium on Distributed Objects and Applications, DOA 2002*, 2002, pp. 1–6. [Online]. Available: http://doc.utwente.nl/55814/

[19] P. Klint, R. Lammel, and C. Verhoef, "Toward an engineering discipline for grammarware," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 14, no. 3, pp. 331–380, 2005.

[20] M. Wimmer and G. Kramler, "Bridging grammarware and modelware," *Proceedings of the Satellite Events at the MoDELS 2005 Conference, Lecture Notes in Computer Science*, vol. 3844, pp. 159–168, 2005.

[21] OMG, "Architecture-driven modernization: Knowledge discovery meta-model (kdm), version 1.3," 2011.

[22] S. Demeyer, S. Ducasse, and S. Tichelaar, "Why unified is not universal? uml shortcomings for coping with round-trip engineering," in *UML'99: The Unified Modeling Language - Beyond the Standard, Second International Conference, Fort Collins, CO, USA, October 28-30, 1999, Proceedings*, 1999, pp. 630–644.

[23] ISO/IEC, "Iso/iec 14977:1996 information technology – syntactic metalanguage – extended bnf," 1996.

[24] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, Eds., *EMF: Eclipse Modeling Framework, 2nd Edition*, 2nd ed. Addison-Wesley Professional, 2008.

[25] J. Earley, "An efficient context-free parsing algorithm," *Communications of the ACM*, vol. 13, no. 2, pp. 94–102, 1970.

[26] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, Eds., *Compilers: Principles, Techniques, and Tools (2nd Edition)*, 2nd ed. Addison Wesley, 2006.

[27] OMG, "Architecture-driven modernization: Abstract syntax tree meta-model (astm), version 1.0," 2009.

[28] R. Kollmann and M. Gogolla, "Capturing dynamic program behaviour with UML collaboration diagrams," in *Fifth Conference on Software Maintenance and Reengineering, CSMR 2001, Lisbon, Portugal, March 14-16, 2001*, 2001, pp. 58–67.

[29] M. Imber, "Case data interchange format standards," *Information and Software Technology*, vol. 33, no. 9, pp. 647–655, 1991.

[30] M. Unterkalmsteiner, R. Feldt, and T. Gorschek, "A taxonomy for requirements engineering and software test alignment," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 2, pp. 16:1–16:38, 2014.

[31] R. L. Glass, "Sorting out software complexity," *Commun. ACM*, vol. 45, no. 11, pp. 19–21, 2002.

[32] B. Kitchenham, O. P. Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman, "Systematic literature reviews in software engineering - a systematic literature review," *Information and Software Technology*, vol. 51, no. 1, pp. 7–15, 2009.

[33] B. Bellay and H. Gall, "A comparison of four reverse engineering tools," in *Proceedings of the 4th Working Conference on Reverse Engineering (WCRE 1997)*. IEEE Computer Society, 1997, pp. 2–11.

[34] M. N. Armstrong and C. Trudeau, "Evaluating architectural extractors," in *Proceedings of the 5th Working Conference on Reverse Engineering (WCRE 1998)*. IEEE Computer Society, 1998, pp. 30–39.

[35] T. C. Lethbridge, "Requirements and proposal for a software information exchange format (sief) standard," http://www.site.uottawa.ca/ tcl/papers/sief/standardProposal.html, 1998.

[36] S. E. Sim, M.-A. Storey, and A. Winter, "A structured demonstration of five program comprehension tools: Lessons learnt," in *Proceedings of the 7th Working Conference on Reverse Engineering (WCRE 2000)*. IEEE Computer Society, 2000, pp. 210–212.

[37] R. Ferenc, S. E. Sim, R. C. Holt, R. Koschke, and T. Gyimothy, "Towards a stardard schema for c/c++," in *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE 2001)*. IEEE Computer Society, 2001, pp. 49–58.

[38] R. Ferenc, Á. Beszédes, M. Tarkiainen, and T. Gyimóthy, "Columbus - reverse engineering tool and schema for C++," in *18th International Conference on Software Maintenance (ICSM 2002), Maintaining Distributed Heterogeneous Systems, 3-6 October 2002, Montreal, Quebec, Canada*, 2002, pp. 172–181.

[39] F. Arcelli, S. Masiero, C. Raibulet, and F. Tisato, "A comparison of reverse engineering tools based on design pattern decomposition," in *Proceedings of the 2005 Australian Software Engineering Conference (ASWEC'05)*. IEEE Computer Society, 2005, pp. 262–269.

[40] C. Amelunxen, A. Königs, and T. Rötschke, "MOSL: composing a visual language for a metamodeling framework," in *2006 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2006), 4-8 September 2006, Brighton, UK*, 2006, pp. 81–84.

[41] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Software Engineering Institute, Tech. Rep. CMU/SEI-90-TR-21, Nov. 1990.

[42] S. R. Tilley, K. Wong, M. D. Storey, and H. A. Müller, "Programmable reverse engineering," *International Journal of Software Engineering and Knowledge Engineering*, vol. 4, no. 4, pp. 501–520, 1994.

[43] G. Saint-Denis, R. Schauer, and R. K. Keller, "Selecting a model interchange format: The SPOOL case study," in *33rd Annual Hawaii International Conference on System Sciences (HICSS-33), 4-7 January, 2000, Maui, Hawaii, USA*, 2000, pp. 1–10.

[44] D. Jin, "Exchange of software representations among reverse engineering tools," Department of Computing and Information Science, Queen's University, Tech. Rep., 2001.

[45] D. Jin, J. Cordy, and T. Dean, "Where's the schema? a taxonomy of patterns for software exchange," in *Proceedings of the 10th International Workshop on Program Comprehension (IWPC 2002)*. IEEE Computer Society, 2002, pp. 65–74.

[46] K. Czarnecki and S. Helsen, "Classification of model transformation approaches," in *OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[47] C. N. Christopher, "Evaluating static analysis frameworks," http://www.cs.cmu.edu/ aldrich/courses/654/tools/christopher-analysis-frameworks-06.pdf, 2006.

[48] H. Wu, "Test case generation for programming language metamodels," in *Proceedings of the 1st Doctoral Symposium of the International Conference on Software Language Engineering (SLE)*, 2010, pp. 27–30.

[49] J. Gray, J. Zhang, Y. Lin, S. Roychoudhury, H. Wu, R. Sudarsan, A. Gokhale, S. Neema, F. Shi, and T. Bapty, "Model-driven program transformation of a large avionics framework," *Proceedings of the 3rd International Conference on Generative Programming and Component Engineering (GPCE'04), Lecture Notes in Computer Science*, vol. 3286, pp. 361–378, 2004.

[50] D. Budgen, A. Burn, O. Brereton, B. Kitchenham, and R. Pretorius, "Empirical evidence about the uml: A systematic literature review," *Software: Practice and Experience*, vol. 41, no. 4, pp. 363–392, 2011.

[51] A. Kunert, "Semi-automatic generation of metamodels and models from grammars and programs," *Electronic Notes in Theoretical Computer Science*, vol. 211, pp. 111–119, 2008.

[52] F. Jouault and J. Bezivin, "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*, vol. 4037, pp. 171–185, 2006.

[53] S. Ducasse, M. Lanza, and S. Tichelaar, "Moose: an extensible language-independent environment for reengineering object-oriented systems," in *2nd International Symposium on Constructing Software Engineering Tools (COSET 2000)*, 2000.

[54] O. Nierstrasz, E. Tichelaar, and S. Demeyer, "Cdif as the interchange format between reengineering," in *In: Proceedings of the OOPSLA Workshop on Model Engineering, Methods and Tools Integration with CDIF*, 1998.

[55] M. van den Brand, H. de Jong, and P. Oliver, "A common exchange format for reengineering tools based on aterms," in *Proceedings of the Workshop on Standard Exchange Formats (WoSEF) at the 22nd International Conference on Software Engineering (ICSE'00)*, 2000.

[56] M. van den Brand and P. Klint, "Aterms for manipulation and exchange of structured data: It's all about sharing," *Information & Software Technology*, vol. 49, no. 1, pp. 55–64, 2007.

[57] J. Czeranski, T. Eisenbarth, H. M. Kienle, R. Koschke, E. Plödereder, D. Simon, Y. Z. V, J. Girard, and M. Würthner, "Data exchange in bauhaus," in *Proceedings of the Seventh Working Conference on Reverse Engineering, WCRE'00, Brisbane, Australia, November 23-25, 2000*, 2000, pp. 293–295.

[58] T. Lin, R. Cheung, Z. He, and K. Smith, "Exploration of data from modeling and simulation through visualization," in *Proceedings of the 3rd International SimTect Conference, Adelaide, Australia*, 1998.

[59] R. Holt, "An introduction to ta: The tuple attribute language," *Department of Computer Science, University of Waterloo and Toronto*, 1998.

[60] S. Lapierre, B. Laguë, and C. Leduc, "Datrix™ source code model and its interchange format: lessons learned and considerations for future work," *ACM SIGSOFT Software Engineering Notes*, vol. 26, no. 1, pp. 53–56, 2001.

[61] A. Schürr, "Developing graphical (software engineering) tools with PROGRES," in *Pulling Together, Proceedings of the 19th International Conference on Software Engineering, Boston, Massachusetts, USA, May 17-23, 1997.*, 1997, pp. 618–619.

[62] J. Ebert, B. Kullbach, and A. Winter, "Grax - an interchange format for reengineering tools," in *Sixth Working Conference on Reverse Engineering, WCRE '99, Atlanta, Georgia, USA, October 6-8, 1999*, 1999, p. 89.

[63] R. C. Holt, A. Schürr, S. E. Sim, and A. Winter, "GXL: A graph-based standard exchange format for reengineering," *Sci. Comput. Program.*, vol. 60, no. 2, pp. 149–170, 2006.

[64] H. M. Kienle and H. A. Müller, "Rigi - an environment for software reverse engineering, exploration, visualization, and redocumentation," *Sci. Comput. Program.*, vol. 75, no. 4, pp. 247–263, 2010.

[65] W3C, "Extensible markup language (xml)," http://www.w3.org/XML/, 2000.

[66] OMG, "Xml metadata interchange (xmi), version 2.5.1," http://www.omg.org/spec/XMI/2.5.1/, 2015.

[67] G. Antoniol, M. D. Penta, and E. Merlo, "Yaab (yet another ast browser): Using ocl to navigate asts." in *Proceedings of the 11th IEEE International Workshop on Program Comprehension (IWPC 2003)*, 2003, pp. 13–22.

[68] A. Bergmayr and M. Wimmer, "Generating metamodels from grammars by chaining translational and by-example techniques," *Proceedings of the First International Workshop on Model-driven Engineering By Example co-located with ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2013), CEUR Workshop Proceedings*, vol. 1104, pp. 22–31, 2013.

[69] V. U. Gómez, S. Ducasse, and T. D'Hondt, "Ring: A unifying meta-model and infrastructure for smalltalk source code analysis tools," *Computer Languages, Systems & Structures*, vol. 38, no. 1, pp. 44–60, 2012.

[70] ISO/IEC, "Iso/iec 25010:2011 systems and software engineering - systems and software qualityrequirements and evaluation (square) - system and software quality models," 2011.

[71] D. Smite, C. Wohlin, Z. Galvina, and R. Prikladnicki, "An empirically based terminology and taxonomy for global software engineering," *Empirical Software Engineering*, vol. 19, no. 1, pp. 105–153, 2014.

[72] L. Martinez, C. Pereira, and L. Favre, "Recovering sequence diagrams from object-oriented code - an ADM approach," in *ENASE 2014 - Proceedings of the 9th International Conference on Evaluation of Novel Approaches to Software Engineering, Lisbon, Portugal, 28-30 April, 2014*, 2014, pp. 188–195.

[73] J. L. C. Izquierdo and J. G. Molina, "An architecture-driven modernization tool for calculating metrics," *IEEE Software*, vol. 27, no. 4, pp. 37–43, 2010.

[74] R. Pérez-Castillo, I. G. R. de Guzmán, R. Gómez-Cornejo, M. Fernández-Ropero, and M. Piattini, "ANDRIU. A technique for migrating graphical user interfaces to android (S)," in *The 25th International Conference on Software Engineering and Knowledge Engineering, Boston, MA, USA, June 27-29, 2013.*, 2013, pp. 516–519.

[75] R. S. Durelli, D. S. M. Santibáñez, M. E. Delamaro, and V. V. de Camargo, "Towards a refactoring catalogue for knowledge discovery metamodel," in *Proceedings of the 15th IEEE International Conference on Information Reuse and Integration, IRI 2014, Redwood City, CA, USA, August 13-15, 2014*, 2014, pp. 569–576.

[76] D. S. M. Santibáñez, R. S. Durelli, and V. V. de Camargo, "A combined approach for concern identification in KDM models," *J. Braz. Comp. Soc.*, vol. 21, no. 1, p. 10, 2015.

[77] S. Tichelaar, S. Ducasse, S. Demeyer, and O. Nierstrasz, "A meta-model for language-independent refactoring," in *Proceedings of the International Symposium on Principles of Software Evolution*, 2000, pp. 154–164.

[78] T. Mens and M. Lanza, "A graph-based metamodel for object-oriented software metrics," *Electr. Notes Theor. Comput. Sci.*, vol. 72, no. 2, pp. 57–68, 2002.

[79] M. Lanza, "Codecrawler - lessons learned in building a software visualization tool," in *7th European Conference on Software Maintenance and Reengineering (CSMR 2003), 26-28 March 2003, Benevento, Italy, Proceedings*, 2003, pp. 409–418.

[80] A. Brühlmann, T. Gîrba, O. Greevy, and O. Nierstrasz, "Enriching reverse engineering with annotations," in *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings*, 2008, pp. 660–674.

[81] V., T. Mahesh, and A. Srivastava, "Performance and language compatibility in software pattern detection," in *IEEE International Advance Computing Conference, 2009 (IACC 2009)*, 2009, pp. 1639–1643.

[82] R. K. Keller, J. Bédard, and G. Saint-Denis, "Design and implementation of a uml-based design repository," in *Advanced Information Systems Engineering, 13th International Conference, CAiSE 2001, Interlaken, Switzerland, June 4-8, 2001, Proceedings*, 2001, pp. 448–464.

[83] M. K. Abdi, H. Lounis, and H. A. Sahraoui, "Analyzing change impact in object-oriented systems," in *32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA 2006), August 29 - September 1, 2006, Cavtat/Dubrovnik, Croatia*, 2006, pp. 310–319.

[84] I. Sora, "A meta-model for representing language-independent primary dependency structures," in *ENASE 2012 - Proceedings of the 7th International Conference on Evaluation of Novel Approaches to Software Engineering, Wroclaw, Poland, 29-30 June, 2012.*, 2012, pp. 65–74.

[85] ——, "Unified modeling of static relationships between program elements," in *Evaluation of Novel Approaches to Software Engineering - 7th International Conference, ENASE 2012, Warsaw, Poland, June 29-30, 2012, Revised Selected Papers*, 2012, pp. 95–109.

[86] T. Gîrba and S. Ducasse, "Modeling history to analyze software evolution," *Journal of Software Maintenance*, vol. 18, no. 3, pp. 207–236, 2006.

[87] V. U. Gómez, A. Kellens, J. Brichau, and T. D'Hondt, "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, Amsterdam, Netherlands, August 24-28, 2009*, 2009, pp. 79–88.

[88] G. Antoniol, M. D. Penta, H. C. Gall, and M. Pinzger, "Towards the integration of versioning systems, bug reports and source code meta-models," *Electr. Notes Theor. Comput. Sci.*, vol. 127, no. 3, pp. 87–99, 2005.

[89] M. Pinzger, H. C. Gall, and M. Fischer, "Towards an integrated view on architecture and its evolution," *Electr. Notes Theor. Comput. Sci.*, vol. 127, no. 3, pp. 183–196, 2005.

[90] E. Engström and K. Petersen, "Mapping software testing practice with software testing research - serp-test taxonomy," in *Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13-17, 2015*, 2015, pp. 1–4.