Patterns for Program Reverse Engineering from the Viewpoint of Metamodel

HIRONORI WASHIZAKI, Waseda University
YANN-GAËL GUÉHÉNEUC, Ecole Polytechnique de Montreal
FOUTSE KHOMH, Ecole Polytechnique de Montreal

Reverse engineering tools often define their own metamodels according to their purposes and intended features. These tools and metamodels have advantages that may benefit other metamodels as well as limitations that other metamodels may solve. To guide practitioners (and researchers) in selecting, integrating, and using appropriate tools, we propose a preliminary pattern catalog for program reverse engineering from the program metamodel viewpoint based on our conceptual framework in consideration of both grammarware and modelware approaches. The catalog consists of one metapattern, **Transformation to higher abstraction levels**, and three concrete patterns, **Integrated program reverse engineering**, **Fact extraction**, and **Architecture recovery**. The intended audience of these patterns is practitioners (and researchers) such as software maintainers who desire to comprehend a program. In addition, these patterns may be helpful for tool developers (and researchers) creating reverse engineering tools.

Categories and Subject Descriptors: D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

General Terms: Languages

Additional Key Words and Phrases: Model-driven engineering, reverse engineering tools, visualization

ACM Reference Format:

Washizaki, H. Guéhéneuc, Y.G. and Khomh, F. 2016. Patterns for Program Reverse-Engineering from the Viewpoint of Metamodel. HILLSIDE Proc. of Conf. on Pattern Lang. of Prog. 22 (October 2015), 10 pages.

1. INTRODUCTION

Because reliable information is often only embedded in the source code when maintaining a software system [Canfora et al. 2011], this paper focuses on *program reverse engineering* (i.e., the process of analyzing the program source code written in general purpose programming languages (GPLs) [Garwick 1968; Buchner and Matthes 2006]) to identify program code elements and to create representations of a program at a certain level of abstraction.

Reverse engineering tools often define their own metamodels according to their purposes and intended features [Ebert et al. 2002]. These tools and metamodels have advantages that may benefit other metamodels as well as limitations that other metamodels may solve. To guide practitioners (and researchers) in selecting, integrating, and employing the appropriate tools, previous works have evaluated, compared, and conducted case studies [Bellay and Gall 1997; Armstrong and Trudeau 1998; Sim et al. 2000; Arcelli et al. 2005; Jin and Cordy 2006;

This work was supported by JSPS KAKENHI Grant Numbers 25330091, 15H02686, 16H02804 and IISF SSR Forum 2015 and 2016.
Author's address: H. Washizaki, 3-4-1 Shinokubo, Shinjuku-ku, Tokyo, 1698555 Japan; email: washizaki@waseda.jp; Y.G. Guéhéneuc, F. Khomh, CP 6079 succ. Centre Ville, Quebec, Canada; email: {yann-gael.gueheneuc, foutse.khomh}@polymtl.ca

Izquierdo and Molina 2014]. Moreover, there are several reverse engineering activity patterns [Demeyer et al. 2000; Demeyer et al. 2002; Murray and Lethbridge 2005; Flores and Aguiar 2008], metapatterns [Favre and NGuyen 2005], exchange patterns [Jin et al. 2002], and a tool's architectural metaphor [Langel et al. 2001] dealing with or related to program reverse engineering. However, to the best of our knowledge, there is no comprehensive set of patterns specific to program reverse engineering from the viewpoint of program metamodels that consider both grammarware [Klint et al. 2005] and modelware approaches [Kurtev et al. 2002]. In the last decade, much effort has been invested to bridge the gap between grammarware and modelware. Consequently, both grammarware and modelware have benefited by transferring artifacts and techniques between these technological spaces [Bergmayr and Wimmer 2013].

We believe that a consistent catalog of related patterns will provide a comprehensive guidance for understanding program reverse engineering and utilizing reverse engineering tools. The goal of this paper is to report the preliminary results of our ongoing work on a consistent catalog of program reverse engineering patterns, which consider both grammarware and modelware approaches. These patterns are intended for practitioners (and researchers) such as software maintainers who want to comprehend a program. In addition, these patterns may be useful for developers (and researchers) creating reverse engineering tools. To realize a consistent catalog, we use our conceptual framework [Washizaki et al. 2016] to explain program metamodels and related concepts as the basis of these patterns.

The remainder of this paper is organized as follows. First, we introduce our conceptual framework in Section 2. In Section 3, we describe our pattern. Finally, we conclude our work and discuss future work in Section 4.

2. TERMINOLOGY AND CONCEPTUAL FRAMEWORK

Program metamodels are used under various contexts such as forward engineering and reverse engineering and at different abstraction levels from architecture to code. However, the concepts associated with a metamodel are not uniformly recognized. To establish a common vocabulary, we designed a conceptual framework [Washizaki et al. 2016]. Figure 1 shows our framework, which adopts the OMG metamodel hierarchy [Kurtev et al. 2002; OMG 2015] with modifications to make it comparable to other model-driven engineering frameworks and views. We define the following concepts:

—*Model*: A simplification of a system built with an intended goal [Favre and NGuyen 2005].

—*Metamodel*: A model of a language that captures the essential properties and features [Clark et al. 2015] of the target models. Although metamodels have primarily been developed and advertised by the OMG with its Meta Object Facility (MOF) [OMG 2015] standard [Alanen and Porres 2003] in the context of modelware, they are not limited to MOF. Examples of metamodels include *Program metamodels* in modelware, *schemas* (or *exchange format*) in dataware, and *grammars* in grammarware [Favre and NGuyen 2005], which are models of program modeling languages, data languages, and programming languages, respectively.
Modelware refers to a technical space using modeling languages and tools [Wimmer and Kramler 2005]. Grammarware is a technical space comprising grammars (i.e., grammar formalisms and grammar notations) and all grammar-dependent software (i.e., software that involves grammar knowledge in an essential manner) [Klint et al. 2005]. Dataware is a technical space dedicated to handling and managing data based on certain schemas; databases are typical examples of dataware.

—*Program metamodel*: A model of programming language grammar, which represents target programs according to a specific purpose. A program model must conform to its program metamodel. Examples of a program metamodel include FAMOOS Information Exchange Model (FAMIX) [Demeyer et al. 1999], Knowledge Discovery Meta-Model (KDM) [OMG 2011], and UML. Figure 2 shows excerpts of FAMIX and KDM.

—*Metalanguage*: A program language to describe program metamodels. Metalanguages can be classified as *metasyntaxes of grammar* such as Extended BNF (EBNF) [ISO/IEC 1996] in textual presentation or *meta-*

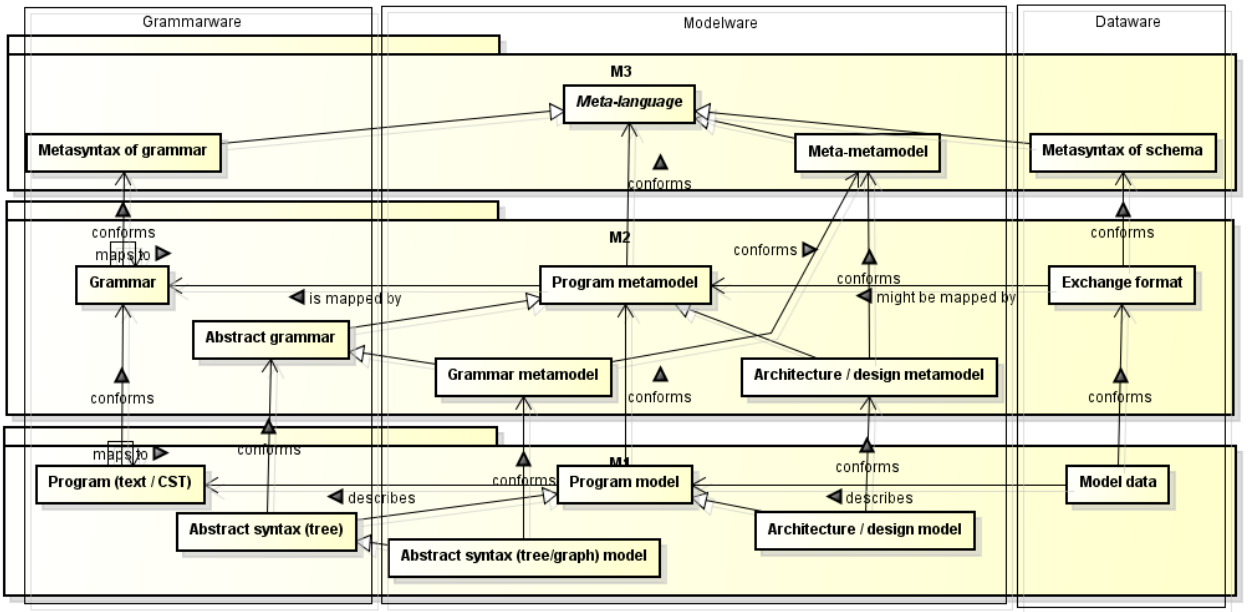

Fig. 1. Conceptual framework of program metamodels

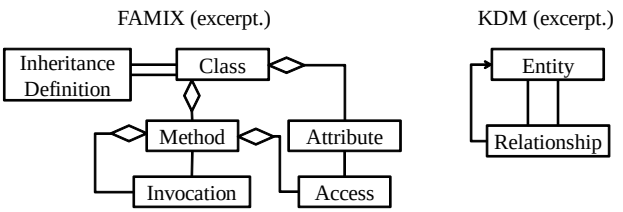

Fig. 2. Examples of program metamodels in the form of UML class diagrams

metamodels of metamodels at certain abstraction levels such as MOF and Eclipse Modeling Framework (EMF) meta model Ecore [Steinberg et al. 2008] usually in a graphic presentation.

—*Context-free grammar* (or simply *grammar*): A formal device to specify which strings are in the language as a set of strings over a finite set of symbols [Earley 1970].

—*Concrete syntax tree (CST)*: A parse tree pictorially depicting how the start symbol of the grammar derives a string in the language [Aho et al. 2006].

—*Abstract syntax tree (AST)*: A simplified syntactic representation of the source code that excludes superficial distinctions of form and unimportant constituents for translation from the tree [Aho et al. 2006]. The AST follows *abstract grammar*, which is a representation of the concrete original grammar at a higher level of abstraction.

—*Abstract syntax model*: A representation of an abstract syntax (tree) in a graphic presentation. Abstract syntax models can be seen as low-level program metamodels. Examples include programming-language-independent AST models such as ASTM [OMG 2009] and programming-language-specific AST models such as Java Metamodel [Kollmann and Gogolla 2001].

—*Standard exchange format (SEF)* (or simply an *exchange format*): A metamodel (i.e., schema) of model data used to store data, which 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), and GraX [Sim and Koschke 2001]. Some of these (e.g., XMI and RDF) are general-purpose exchange formats that can be adapted to software.

3. PATTERNS FOR PROGRAM REVERSE ENGINEERING

Based on the aforementioned conceptual framework, we describe a preliminary pattern catalog for program reverse engineering techniques from the viewpoint of metamodels. Our catalog consists of one metapattern, which is common in reverse engineering, and three concrete patterns realizing the metapattern according to specific contexts. Each 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* together with examples. Moreover, these concrete patterns specify necessary transformations to be used in its own solution as *Transformations*. Figure 3 shows relationships among these patterns.

3.1 Metapattern: Transformation to higher abstraction levels

Program reverse engineering consists of various transformations such as extraction and abstraction. The metapattern **Transformation to higher abstraction levels** describes a common fundamental process of software transformation in any reverse engineering activity. Moreover, the metapattern gives succeeding patterns a common context, problem, and solution. By referring to this metapattern, practitioners and researchers can recognize when, why, and how to perform reverse engineering.

—*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 certain interests 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 4) 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. For example, the abstract grammar of Java and FAMIX can be regarded as a *Lower-meta* and a *Higher-meta*, respectively, when maintainers transform Java source code to FAMIX models. Figure 4 shows the elements involved in the transformation.
Concrete transformations can be classified into four types: *Extraction*, *Abstraction*, *View* and *Store*.

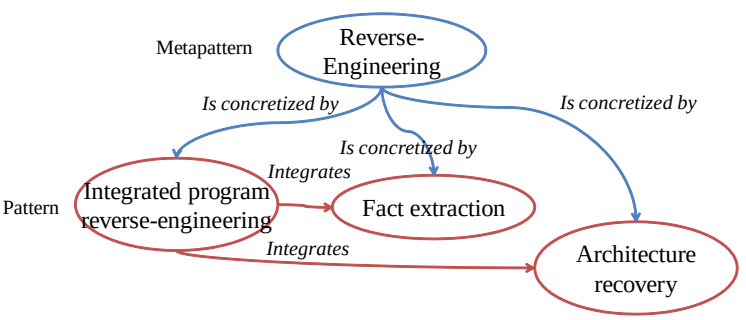

Fig. 3. Relationships among patterns

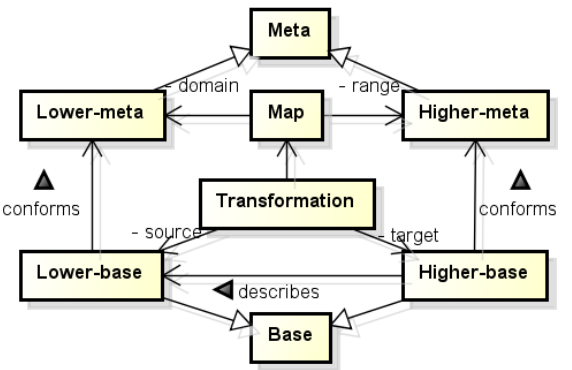

Fig. 4. Structure of **Transformation to higher abstraction levels**

—*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.

3.2 Pattern: Integrated program reverse engineering

—*Alias*: Extract-Abstract-View metaphor [Langel et al. 2001],

—*Transformations*: *Extraction*, *Abstraction*, *Store* and *View*

—*Context*: You are analyzing a program to comprehend and maintain it without employing reverse engineering tools or modelware. Similar comprehension activities are anticipated in the future.

—*Problem*: The description of the program source code itself is too complicated to specify a certain structure, behavior, or concern. There is no reliable description at a higher level of abstraction, which exactly corresponds to the source code.

—*Solution*: This is a two-step solution.
(1) First, prepare the following:
—The grammar of the programming language of the program source code to be analyzed
—Program metamodels for extraction, abstraction, and visual representation
—An exchange format

—Rules mapping among the grammar, the program metamodels, and the exchange format

(2) Then automate the following tasks. Figure 5 shows the elements of the pattern with corresponding roles taken from the metapattern shown in Figure 4.
   (a) Parse and extract the necessary facts from the target program source code (*Extraction*).
   (b) Abstract facts into models at higher abstraction levels if necessary (*Abstraction*). It should be noted that abstraction in addition to fact extraction is not necessary for all reverse engineering activities.
   (c) Store data into a repository (*Store*).
   (d) Transform the program/model data into visual representations and display them for further analysis (*View*).

This solution can be seen as an integration of the following two patterns. The first two tasks can be realized by **Fact extraction** while the remaining two can be realized by **Architecture recovery**.

—*Known implementation*: Any integrated reverse engineering tools such as Bookshelf, DALI, PBS, SoftANAL, SWAGKIT, GUPRO [Ebert et al. 2002], Fujaba, and MOOSE [Ducasse et al. 2000]. For example, GUPRO [Langel et al. 2001] parses Java, C/C++, and IDL source codes to extract program facts with respect to a conceptual model consisting of Module, Class, Method, and Attribute (*Extraction*). The extracted data is stored in a relational database (*Store*). GUPRO supports an ad hoc query using standard SQL statements to select a particular data set such as a set of method pairs of a caller and a callee, stored in the repository (*Abstraction*). Finally, the query results are displayed as HTML tables (*View*).

3.3 Pattern: Fact extraction

—*Alias*: Bridging grammarware and modelware

—*Transformations*: *Extraction* and *Store*

—*Context*: You are using or developing modelware such as UML modeling tools, model-driven (reverse) engineering tools, architecture-driven modernization tools, or visualization tools such as DaVinci and GraphViz. These tools are designed to accept low-level program facts (such as AST) by conforming to certain program metamodels such as FAMIX, ASTM, KDM, and EMF-based program metamodels.

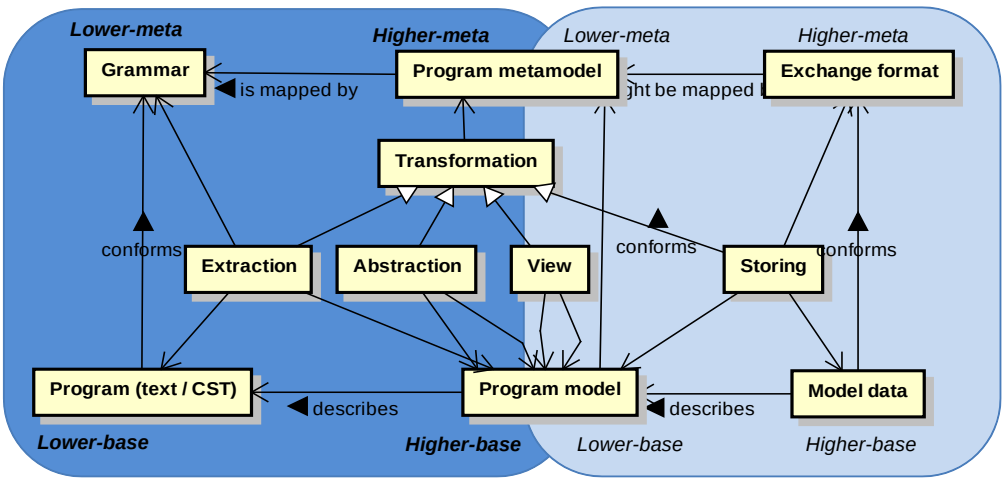

Fig. 5. Structure of **Integrated progrm reverse engineering**

—*Problem*: The available or planned tools cannot directly extract program models from the program source codes written in the programming language (e.g., Java) that you are using. Modeling manually is often unreliable and unscalable for the large-scale complex source codes.

—*Solution*: This is a two-step process.

(1) First, prepare the following:
  —The grammar of the programming language
  —A program metamodel and an exchange format that are acceptable for the available (or planned) tools
  —Rules mapping among the grammar, the metamodel, and the exchange format

(2) Second, automate the following tasks:
  (a) Parse and extract the necessary low-level facts conforming to the metamodel from the target program source code written in the programming language that you are using (*Extraction*).
  (b) Store fact data into a repository (*Store*).

—*Known implementation*: Any fact extractors such as Ccia, cppx, Rigi C++ parser, TkSee/SN [Sim et al. 2002], Datrix and Columbus [Ferenc et al. 2001] for C++, and MOOSE, SPOON, MoDisco, JaMoPP, Stratego/XT and Gra2MoL for other languages, including Java [Heidenreich et al. 2010; Izquierdo and Molina 2014]. These extractors can be roughly classified as dedicated parsers or query languages [Izquierdo and Molina 2014]. For example, JaMoPP [Heidenreich et al. 2010] parses a set of Java source code files, extracts program facts such as Java classes and methods with respect to an EMF-based Java metamodel (*Extraction*), and exports the extracted data in the compact serialization format (*Store*).

3.4 Pattern: Architecture recovery

—*Alias*: Design recovery

—*Transformations*: *Extraction*, *Abstraction* and *View*

—*Context*: You are analyzing a program to capture its high level design/architecture but are not using or developing reverse engineering tools or modelware.

—*Problem*: Because there is no explicit document or model to describe the specific micro-architecture of the program, it is hard to grasp the entire structure and behavior at higher abstraction levels. Drawing entire architecture manually is often unreliable and inconsistent to be maintained on a long-term basis.

—*Solution*: There is a two-step solution.

(1) First, prepare the following:
  —The grammar of the programming language of the program source code to be analyzed
  —Program metamodels for extraction, abstraction, and visual representation
  —Rules mapping among the grammar and the program metamodels

(2) Second, automate the followings tasks.
  (a) Extract necessary facts from the target code (*Extraction*).
  (b) Abstract facts into models at higher abstraction levels (*Abstraction*).
  (c) Transform model data into visual representations and display them (*View*).

—*Known implementation*: Any architecture/design recovery tool such as Rigi. Moreover, design pattern detection tools can be regarded as a kind of architecture recovery tools since the detected design patterns are key elements of software design. For example, the first author developed an automatic component-extraction system targeting Java source code [Washizaki and Fukazawa 2005], which parses the Java source code, extracts AST models with respect to the Java grammar metamodel (*Extraction*), and selects only basic structural data such as classes (*Abstraction*), methods, fields, and dependencies among them with respect to a class relation graph (CRG) as the metamodel. Finally, the system displays the results in the form of a UML class diagram with information on extractable components (*View*) (Figure 6).

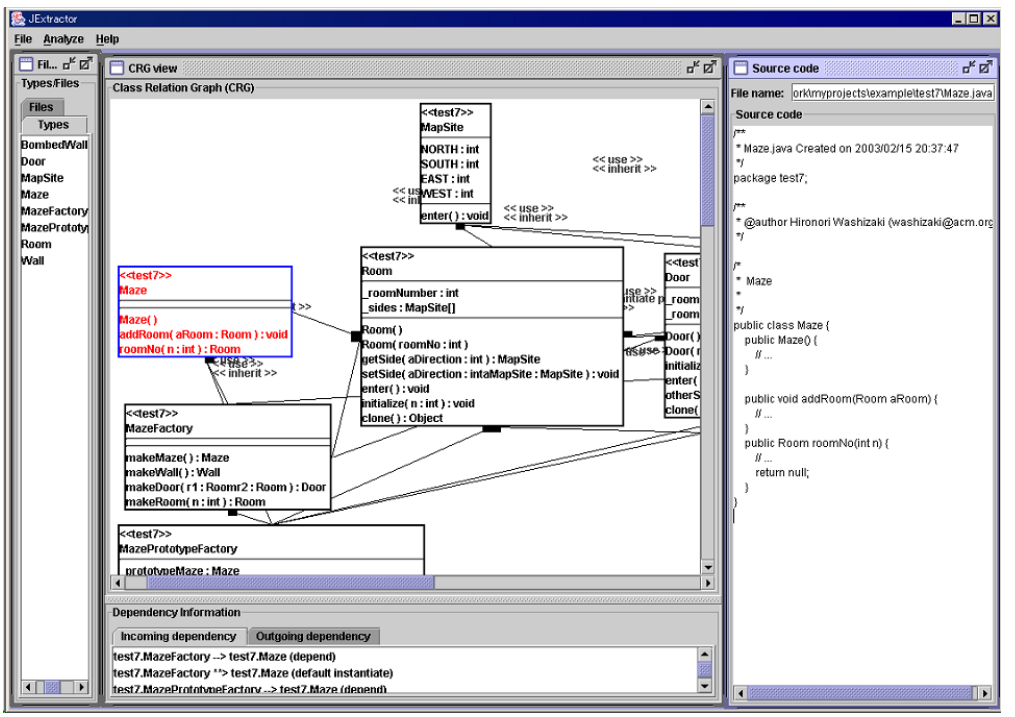

Fig. 6. Example of a view of the component-extraction system

4. CONCLUSION AND FUTURE WORK

Herein our preliminary catalog of program reverse engineering patterns is proposed from the metamodel viewpoint based on our conceptual framework in consideration of both grammarware and modelware approaches. In the future, we plan to extend the catalog to incorporate more reverse engineering (and possibly reengineering) patterns. For example, there could be an additional pattern, **Model-based architecture recovery**, which performs architecture recovery without any low-level fact extraction and it only accepts program models as input. Moreover, we plan to explain concrete transformations (i.e., *Extraction*, *Abstraction*, *View* and *Store*) in detail in the form of additional metapatterns.

Acknowledgement

We thank our shepherd Robert Hanmer for his valuable comments, which significantly improved our paper. Moreover, we thank all participants of the writers' workshop held at the 2016 Conference on Pattern Languages of Programs (PLoP) for their valuable comments and discussions. This work has been conducted as a part of "Research Initiative on Advanced Software Engineering in 2015" supported by Software Reliability Enhancement Center (SEC), Information Technology Promotion Agency Japan (IPA). Moreover, this work was partially supported by JSPS KAKENHI Grant Number 16H02804, IISF SSR Forum 2015 and 2016.

REFERENCES

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman (Eds.). 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition)* (2 ed.). Addison Wesley.

Marcus Alanen and Ivan Porres. 2003. A Relation Between Context-Free Grammars and Meta Object Facility Metamodels. *TUCS Technical Report, No.606* (2003).

Francesca Arcelli, Stefano Masiero, Claudia Raibulet, and Francesco Tisato. 2005. 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, 262–269.

M. N. Armstrong and C. Trudeau. 1998. Evaluating Architectural Extractors. In *Proceedings of the 5th Working Conference on Reverse Engineering (WCRE 1998)*. IEEE Computer Society, 30–39.

Bemdt Bellay and Harald Gall. 1997. A comparison of four reverse engineering tools. In *Proceedings of the 4th Working Conference on Reverse Engineering (WCRE 1997)*. IEEE Computer Society, 2–11.

Alexander Bergmayr and Manuel Wimmer. 2013. 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* 1104 (2013), 22–31.

Thomas Buchner and Florian Matthes. 2006. Introspective Model-Driven Development. *Third European Workshop on Software Architecture (EWSA 2006), Revised Selected Papers, Lecture Notes in Computer Science* 4344 (2006), 33–49.

Gerardo Canfora, Massimiliano Di Penta, and Luigi Cerulo. 2011. Achievements and challenges in software reverse engineering. *Commun. ACM* 54, 4 (2011), 142–151.

Tony Clark, Paul Sammut, and James Willans. 2015. Applied Metamodelling: A Foundation for Language Driven Development (Third Edition). *ArXiv e-prints* (2015).

Serge Demeyer, Stephane Ducasse, and Oscar Nierstrasz. 2000. A Pattern Language for Reverse Engineering. In *Proceedings of the 5th European Conference on Pattern Languages of Programs (EuroPLoP'2000)*. 189–208.

Serge Demeyer, Stephane Ducasse, and Oscar Nierstrasz (Eds.). 2002. *Object-Oriented Reengineering Patterns* (1 ed.). Morgan Kaufmann.

Serge Demeyer, Stéphane Ducasse, and Sander Tichelaar. 1999. 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*. 630–644.

Stephane Ducasse, Michele Lanza, and Sander Tichelaar. 2000. MOOSE: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems. In *2nd International Symposium on Constructing Software Engineering Tools (COSET 2000)*.

Jay Earley. 1970. An Efficient Context-Free Parsing Algorithm. *Commun. ACM* 13, 2 (1970), 94–102.

Jurgen Ebert, Bernt Kullbach, Volker Riediger, and Andreas Winter. 2002. GUPRO - Generic Understanding of Programs, An Overview. *Electronic Notes in Theoretical Computer Science* 72, 2 (2002), 47–56.

Jean-Marie Favre and Tam NGuyen. 2005. 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* 127, 3 (2005), 59–74.

Rudolf Ferenc, Susan Elliott Sim, Richard C. Holt, Rainer Koschke, and Tibor Gyimothy. 2001. Towards a Stardard Schema for C/C++. In *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE 2001)*. IEEE Computer Society, 49–58.

Nuno Flores and Ademar Aguiar. 2008. Patterns for Understanding Frameworks. In *Proceedings of the 15th Conference on Pattern Languages of Programs (PLoP'08)*. ACM, 1–11.

Jan V. Garwick. 1968. Programming Languages: GPL, a truly general purpose language. *Commun. ACM* 11, 9 (1968), 634–638.

Florian Heidenreich, Jendrik Johannes, Mirko Seifert, and Christian Wende. 2010. Closing the Gap between Modelling and Java. *Proceedings of the International Conference on Software Language Engineering (SLE 2009), Lecture Notes in Computer Science* 5969 (2010), 374–383.

ISO/IEC. 1996. ISO/IEC 14977:1996 Information technology – Syntactic metalanguage – Extended BNF. (1996).

Javier Luis Canovas Izquierdo and Jesus Garcia Molina. 2014. Extracting models from source code in software modernization. *Software and Systems Modeling* 13, 2 (2014), 713–734.

D. Jin, J.R. Cordy, and T.R. Dean. 2002. 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, 65–74.

Dean Jin and James R. Cordy. 2006. Integrating Reverse Engineering Tools Using a Service-Sharing Methodology. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*. IEEE Computer Society, 94–99.

Paul Klint, Ralf Lammel, and Chris Verhoef. 2005. Toward an engineering discipline for grammarware. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 14, 3 (2005), 331–380.

Ralf Kollmann and Martin Gogolla. 2001. Capturing Dynamic Program Behaviour with UML Collaboration Diagrams. In *Fifth Conference on Software Maintenance and Reengineering, CSMR 2001, Lisbon, Portugal, March 14-16, 2001*. 58–67.

Ivan Kurtev, Jean Bézivin, and Mehmet Aksit. 2002. Technological Spaces: An Initial Appraisal. In *International Symposium on Distributed Objects and Applications, DOA 2002*. 1–6. `http://doc.utwente.nl/55814/`

Carola Langel, Harry M. Sneed, and Andreas Winter. 2001. Comparing Graph-based Program Comprehension Tools to Relational Database-based Tools. In *Proceedings of the 9th International Workshop on Program Comprehension (IWPC 2001)*. IEEE Computer Society, 209–218.

Adam Murray and Timothy C. Lethbridge. 2005. Presenting Micro-Theories of Program Comprehension in Pattern Form. In *Proceedings of the 13th International Workshop on Program Comprehension (IWPC'05)*. IEEE Computer Society, 45–54.

OMG. 2009. Architecture-driven Modernization: Abstract Syntax Tree Metamodel (ASTM), Version 1.0. (2009).

OMG. 2011. Architecture-Driven Modernization: Knowledge Discovery Meta-Model (KDM), Version 1.3. (2011).

OMG. 2015. OMG Meta Object Facility (MOF) Core Specification, Version 2.5. (2015).

Susan Elliott Sim, Richard C. Holt, and Steve Easterbrook. 2002. On Using a Benchmark to Evaluate C++ Extractors. In *Proceedings of the 10th International Workshop on Program Comprehension (IWPC'02)*. IEEE Computer Society, 114–123.

Susan Elliott Sim and Rainer Koschke. 2001. WoSEF: Workshop on Standard Exchange Format. *ACM SIGSOFT Software Engineering Notes* 26, 1 (2001), 44–49.

Susan Elliott Sim, Margaret-Anne Storey, and Andreas Winter. 2000. 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, 210–212.

Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks (Eds.). 2008. *EMF: Eclipse Modeling Framework, 2nd Edition* (2 ed.). Addison-Wesley Professional.

Hironori Washizaki and Yoshiaki Fukazawa. 2005. A Technique for Automatic Component Extraction from Object-Oriented Programs by Refactoring. *Science of Computer Programming* 56, 1–2 (2005), 99–116.

Hironori Washizaki, Yann-Gaël Guéhéneuc, and Foutse Khomh. 2016. A Taxonomy for Program Metamodels in Program Reverse Engineering. In *Proceedings of the 32nd IEEE International Conference on Software Maintenance and Evolution (ICSME), October 2-10, Raleigh, North Carolina, USA*. 1–12.

Manuel Wimmer and Gerhard Kramler. 2005. Bridging Grammarware and Modelware. *Proceedings of the Satellite Events at the MoDELS 2005 Conference, Lecture Notes in Computer Science* 3844 (2005), 159–168.

Received June 2016; revised September 2016; accepted February 2017