Frontiers of Reverse Engineering: a Conceptual Model

Gerardo Canfora and Massimiliano Di Penta
RCOST - University of Sannio, Benevento, Italy
canfora@unisannio.it dipenta@unisannio.it

Abstract

Software reverse engineering is a crucial task to reconstruct high-level views of a software system—with the purpose of understanding and/or maintaining it—when the only reliable source of information is the source code, or even the system binaries.

This paper discusses key reverse engineering concepts through a UML conceptual model. Specifically, the model is composed of a set of UML class diagrams describing relationships existing among reverse engineering processes, tools, artifacts, and stakeholders.

1 Introduction

An important aspect of software maintenance is the comprehension of a system to be maintained. It has been estimated that over 50% of the maintenance effort is due to program comprehension [51]. Comprehending a software system to perform a maintenance task requires to access different kinds of documentation, ranging from user documentation to analysis and design models. Unfortunately, very often such a documentation is either unavailable or incomplete and inconsistent. As a matter of fact, the only reliable source of information is the system source code or, in some cases—when also the source code is not available—the system binaries. The IEEE-1219 [32] standard recommends reverse engineering as a key supporting technology to deal with systems that have the source code as the only reliable representation. Reverse engineering goals are multiple, e.g., coping with complexity, generating alternate views, recovering lost information, detecting side effects, synthesizing higher abstractions, and facilitating reuse.

The term *reverse engineering* derives from a well-established practice in the field of hardware analysis, where the objective is recovering hardware design with the aim of replicating or modifying it. Rekoff [47] defined reverse engineering as *the process of developing a set of specifications for a complex hardware system by an orderly examination of specimens of that system*. Much in the same way, in many contexts the term *software reverse engineering* relates to recovering source code from a software system binaries. Binary reverse engineering can be done with malicious objectives, e.g., removing software protections or limitations, allowing unauthorized accesses to system/data, but also licit objectives, such as allowing maintenance of systems for which the source code was lost, or assessing a software system security level. This view of reverse engineering will not be further discussed in this paper; more details can be found in a book by Eilam [24].

In the context of software engineering, the most well-known definition of the term *reverse engineering* was provided in 1990 by Chikofsky and Cross [15]: *the process of analyzing a subject system to (i) identify the system's components and their inter-relationships and (ii) create representations of the system in another form or at a higher level of abstraction*. Thus, the core of reverse engineering consists of deriving information from the available software artifacts and translating it into abstract representations more easily understandable by humans. Of course, the benefits are maximal when reverse engineering is supported by tools.

Reverse engineering is a process of examination, not a process of change, and therefore it does not involve changing the software under examination. A related term, which entails changes, is reengineering, defined as [15]: *the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form*.

This paper proposes a conceptual model aimed at describing key processes, tools, artifacts and stakeholders involved in reengineering and reverse engineering activities, pointing, for each process/tool, to key references in the literature. The paper does not aim to provide an exhaustive survey of the topic nor a complete ontology, but rather a way to gain a general understanding of key reverse engineering terms and concepts. A broad survey—also containing a future perspective of reverse engineering—can be found in a different paper from the same authors [11] or in a earlier paper by Müller *et al.* [45], while a deeper description of some reverse engineering-related activities—

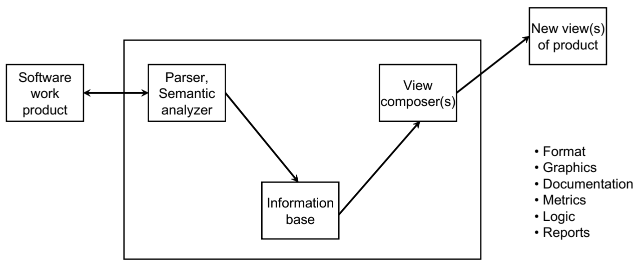

Figure 1. Reverse engineering tools architecture [15]

such as program comprehension, software transformation, clone detection, impact analysis, software visualization, slicing, mining software repositories, and software analysis for security—can be found in other, specific papers of the Frontiers of Software Maintenance volume.

The paper is organized as follows. Section 2, starting from the Chikofsky and Cross definitions, presents the core conceptual model of reverse engineering. Artifacts representing reverse engineering inputs, outputs and intermediate results are described in Section 3. Section 4 provides a taxonomy of different reverse engineering analyses, while Section 5 describes how different software views can be built from the results of analysis. Section 6 shows how the proposed conceptual model can be instantiated to describe a software understanding task performed by using *Moose* [22] and *CodeCrawler* [36]. Finally, Section 7 concludes the paper.

2 The Reverse Engineering Core Model

In their seminal paper, Chikofsky and Cross [15] viewed reverse engineering as a two-step process, composed of two phases, information extraction and abstraction. Information extraction analyzes the subject system artifacts to gather raw data, whereas abstraction creates user-oriented documents and views. For example, information extraction activities consist of extracting Control Flow Graphs (CFGs), metrics, or facts from source code. Abstraction outputs can be design artifacts, traceability links, or business objects. Accordingly, Chikofsky and Cross outlined a basic architecture for reverse engineering tools (cf. Figure 1). The software product to be "reversed" is analyzed, and the results of this analysis are stored into an information base. Such information is then used by view composers to produce alternate views of the software product, such as metrics, graphics, reports, etc.

Stemming from the Chikofsky and Cross idea, we model key concepts of reverse engineering around a "core" model, which can be thought of as an explosion of the architecture of Figure 1. The core model is shown in Figure 2; the gray area maps onto the Chikofsky and Cross reverse engineering tools architecture.

As described in the model, reverse engineering activities are performed to satisfy a *Reverse Engineering goal*, expressed by a *Software Engineer* to solve some specific problems related to a *Software Product*, in particular to a *Legacy Product*, i.e., an old software product that is still being used because it represents a valuable asset for a particular business. Specifically, a reverse engineering goal can be a *Change Goal*, aimed a producing a *Renewed Product*. Examples of change goals include migrating an existing product [8] towards a new platform—such as client-server [10, 50], Web [3] or service oriented architectures [12]—modernizing user interfaces [40, 44], remodularizing a software product by identifying components with clustering techniques [55], formal concept analysis [49], or search-based techniques [41], refactoring clones, removing dead code, and reducing application footprint with the purpose of porting the system towards resource limited devices [20].

A reverse engineering goal can also be an *Understanding Goal*, aimed at increasing the current understanding of a software product, building high-level software artifacts where they are not available, or in general building high-level *Software Views*. This can be performed at different levels, starting from recovering code from binary when it is not available [16], to recovering architectures [34], UML diagrams [54], requirements [58], and traceability links between source code and high-level documentation [1, 38].

High-level views are produced by mean of an *Abstractor*. The abstractor creates the views by using the information extracted from software artifacts (composing the software product) and stored in the *Information Base*. The information base is a repository used to store information extracted by *Analyzers*. Such a repository can be internal to a particular tool, thus not accessible from the outside, or it can be queried by other tools. Examples of information bases include: the repository of the *CIA* environment [14], or the object-oriented database (*Refine*) used by the the *Software Refinery* toolkit [39] to store a fine-grained program model in the form of an attributed Abstract Syntax Tree (AST). The *Tuple Attribute* (TA) language [29] represents facts related to programs—although applications of TA are not limited to programs—as a set of binary relations. When designing an information base, it is desirable to do it using a common schema shared by different tools. Efforts in this direction produced schema such as the *Graph eXchange Language* (GXL) [56, 30], the *Rigi Standard Format* (RSF) [57], or the *FAMIX* metamodel used by *Moose* [22].

Recently, the Object Management Group (OMG) foundation for software modernization has proposed the Knowledge Discovery Metamodel (KDM)[1], a common intermedi-

[1] *http://www.omg.org/technology/documents/modernization_spec_catalog.htm*

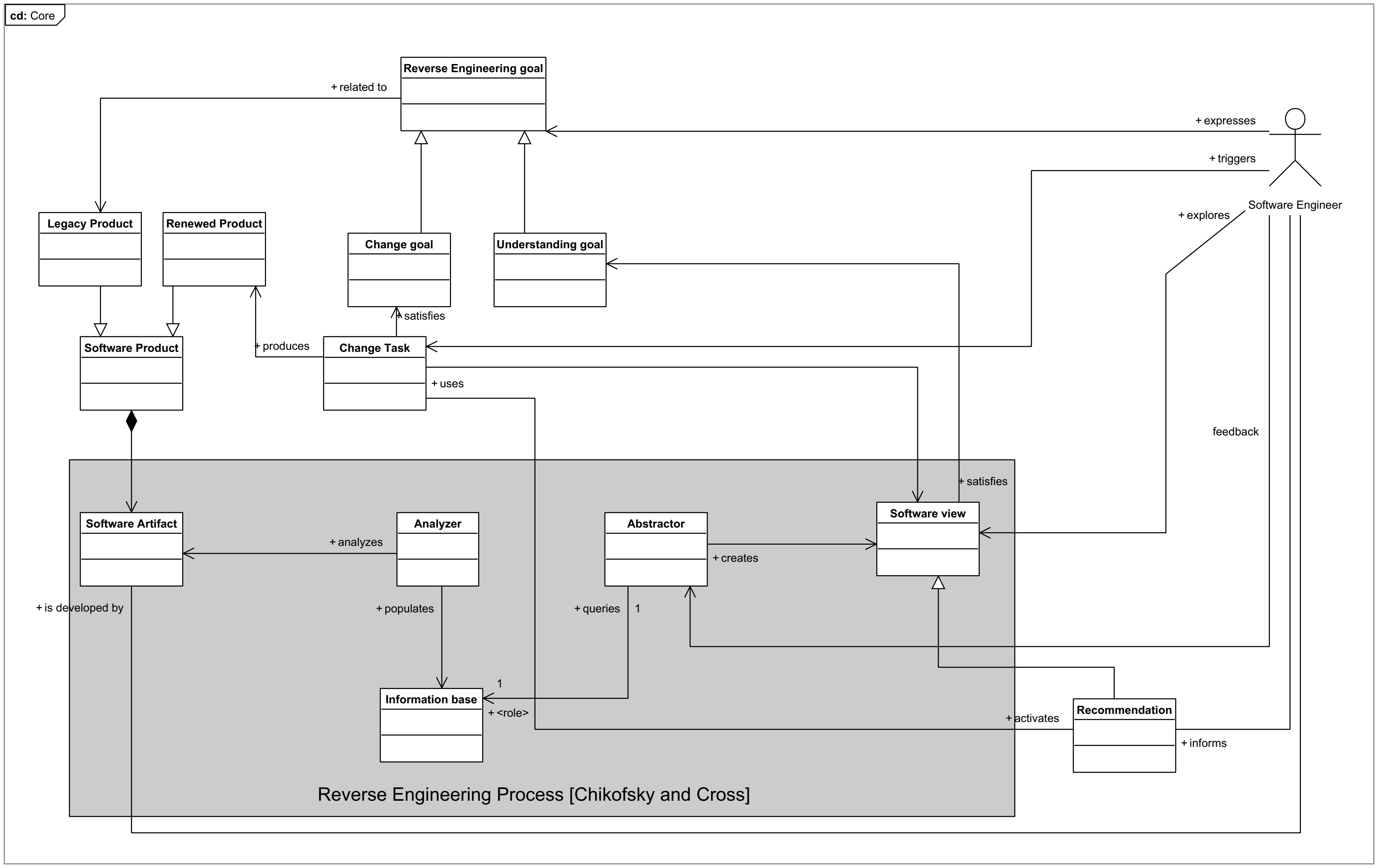

Figure 2. Reverse engineering core model

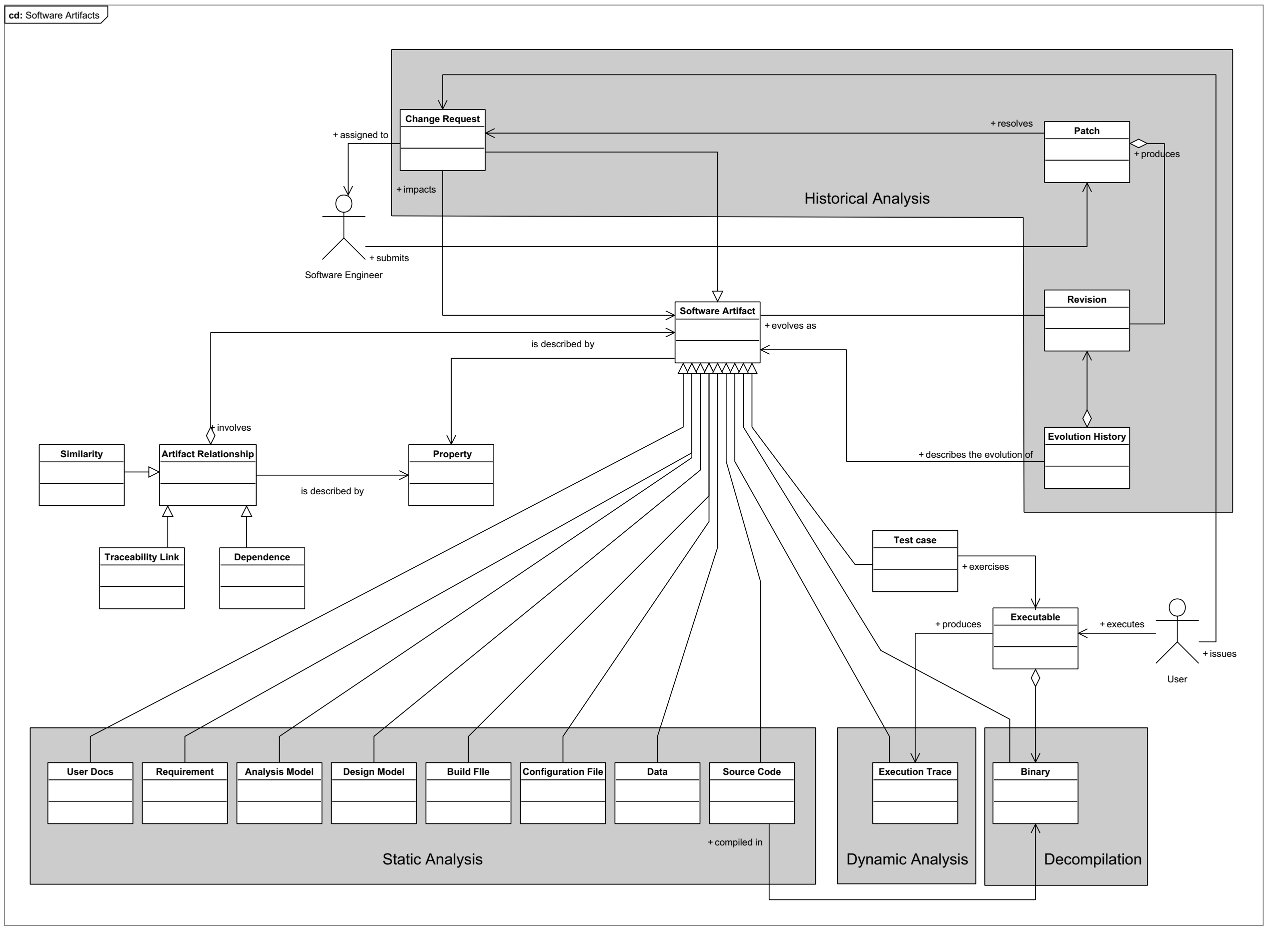

Figure 3. Reverse engineering artifacts

ate representation describing aspects of knowledge related to different aspects (and artifacts) involved in software development or in modernization activities. KDM allows for representing software entities at different levels of granularity by using the notion of *container*, i.e., an entity owns other entities. Also, it provides a precise—language and platform independent—semantic by means of a low-level model, called *micro-KDM.*

A software view can be directly explored by software engineers that use it to satisfy their understanding goal. In addition, the software engineer can provide feedbacks to the reverse engineering tool with the aim of producing refined, more precise, views. An example of feedback—based on the Rocchio Vector Space Model feedback mechanism—was used to improve the performances of traceability recovery [31]. Experts' feedbacks were used to improve the results of an automatic source code remodularization [20]. A particular case of *Software View* is a *Recommendation system* [48]: based on the information available in the view, a recommendation system provides suggestions to the software engineer, and if needed activates a change task. Recommendation systems are useful because, very often, views provide the software engineer with a large amount of information s/he is not able to benefit [46]. Examples of recommendation systems include tools for keeping consistency in cloned code being maintained [21], tools suggesting refactoring opportunities or tools suggesting the most appropriate developer for a maintenance task [2].

The next sections will detail specific concepts expressed in the core conceptual model, in particular concepts related to artifacts, analyzers, and views.

3 Reverse Engineering Artifacts

Reverse engineering aims at building and analyzing views for different kinds of *Software Artifacts*. Figure 3 provides a non-exhaustive classification of some possible software artifacts. The figure also highlights artifacts that can be used for static analysis (*User Documentation*, *Re-*

quirements, *Analysis Models*, *Design Models*, *Configuration Files*, *Build files*, *Source Code*, *Data*), dynamic analysis (*Execution Traces*, obtained by exercising *Executables* with test cases or in real usage scenarios, where end-users interact with the application), decompilation (*Binaries*), and historical analysis (*Change Requests*, *Patches*, and sequences of file *Revisions* describing an artifact *Evolution History*).

Each software artifact can be characterized by a set of *Properties*, such as metrics, authorship information, effort needed to produce the artifact, etc. There exist a series of *Relationships* between artifacts, for example *Similarity*, possibly indicating the presence of clones, *Dependences*, such as coupling or data flow dependences, and *Traceability Links*.

From an historical perspective, the *Evolution History* of a software artifact can be viewed as a sequence of *Revisions* managed by a versioning tool, e.g., Concurrent Versions Systems (CVS)[2] or SubVersioN (SVN)[3]. Very often, artifact revisions are produced by a *Patch* submitted by a software engineer to fulfill a *Change Request* issued by a system user. A change request may describe a failure occurred in the system—and thus is solved by means of a bug fixing— or a proposal of enhancement; in both cases it impacts a number of *Software Artifacts*.

Ideally, all the above mentioned artifacts and relationships should be available to understand a software system. As mentioned in the introduction, however, not all of them are available and, in many cases, a maintainer can only rely on the source code or, even worse, on binaries. Reverse engineering will have therefore a multiple role:

- reconstructing artifacts that are either unavailable—e.g., extracting design or analysis model from source code or source code from binaries—or unaligned with the rest of the system, e.g., requirements or design models that are inconsistent with respect to the source code;

- building other high-level views, aimed at providing the software engineer with information/insights necessary to understand a software system and, if necessary, to perform change tasks.

- provide the information needed to perform software change tasks in a model-driven reengineering scenario. The horseshoe model by Kazman *et al.* [33] foresees reengineering as a three-steps process: the first step aims at extracting the architecture from source code; the second step transforms the extracted architecture towards a target architecture; finally, the third step instantiates the new architecture by generating new source code.

[2] *http://www.nongnu.org/cvs/*

[3] *http://subversion.tigris.org/*

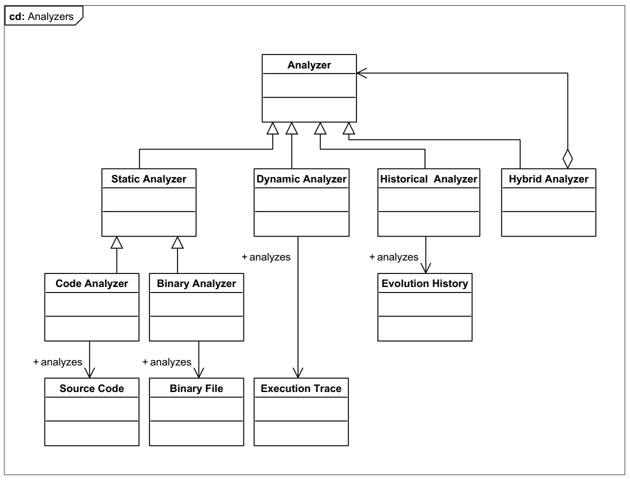

Figure 4. Software Analyzers

4 Software Analyzers

Figure 4 describes the role of *Analyzers* in reverse engineering activities, and how they can be combined to perform complex reverse engineering tasks. Analyzers extract information from a software artifact, with the purpose of populating the *Information Base*. The most common kind of software analyzer is the *Static Analyzer*, which extracts information from software a artifact by processing, in some ways, the file containing the software artifact itself.

A particular case of static analyzer is the *Code Analyzer*, that extracts information from the source code. This can happen in different ways: some extractors first build the source code AST and populate the information base with the AST itself. Examples of analysis tools are the *Design Maintenance System (DMS)* [4], TXL [18], and the *Stratego* toolkit [7]. The above mentioned tools are more then simply analyzers, in that they also provide transformation facilities. In particular, TXL is merely a source-to-source transformation engine, that allows for performing software analyses by transforming the source code, e.g., the transformation produces a high-level view, an instrumented code to be used for performing dynamic analysis, or a fact to be stored in the information base.

Other analyzers extract relevant information—e.g., metrics—from software artifacts and populate an information base. Examples of this kind of analyzers are *Columbus* [27], *CPPX* [19]—which modifies the *gcc* compiler for fact extration purposes and populates a TA information base—or the *Bauhaus* fact extractor [34]. SrcML [17] is a source code analyzer that parses several languages and produces an XML-encoded AST.

Not all the extractors require to parse the source code completely. This is the case of island and lake parsers. Island parsers only parse source code fragments of interest, ignoring any other token and activating the parser only when specific tokens are encountered. For example, if one is interested to analyze embedded SQL, all tokens until the "BEGIN SQL" is encountered are ignored, then the parser is activated until the "END SQL" token is found. Lake parsers, instead, are able to ignore source code fragments not contemplated by the grammar, e.g., related to programming language dialects. Moonen [43] developed the idea of source code analysis through island parsing and lake parsing, to cope with source code analysis difficulties due to the diffusion of a wide number of programming languages dialects—a phenomenon known as the "500 language problem" [35]—otherwise requiring the availability of many robust grammars.

Static analyzers are not limited to source code analyzers, there are analyzers for binary files [16, 24], but also for design models and requirements. This is the case, for instance, of extractors aimed at indexing requirement text for traceability recovery purposes [1]. Static analysis is reasonably fast, precise and cheap. However, many peculiarities of programming languages, such as the presence of pointers, or of constructs such as polymorphism, make static analysis difficult and sometimes imprecise. In addition, the extraction of some specific information related, for instance, to the interaction of an user with the application or to the sequence of messages exchanged between objects, is difficult, if not infeasible, to perform statically. Nowadays, mechanisms such as the *reflection*—provided by some programming languages such as Java—can ease some analysis tasks, for example providing access to fields and methods of a given class.

To overcome the limitations of static analysis, reverse engineers can count on *Dynamic Analysis*, aimed at extracting information from *Execution traces*. Dynamic analysis, on its side, can be helpful to deal with polymorphic invocations, does not need to resolve pointers, and is more suited to capture the behavior of a program. For example, it has been used by Ernst to detect program likely invariants [26]. On the other hand, dynamic analysis requires the system to be compiled, which may not be necessarily true when one wants to analyze a system snapshot downloaded from a versioning system. Above all, results of dynamic analysis strongly depend on the quality of input data used to exercise the program. To extract execution traces, program instrumentation may be required, although this is not always necessary: analysis facilities provided by the Java Virtual Machine (JVM^{TM}) 1.5 on, through the JVM Tool Interface (JVMTI), allows to get dynamic information directly from the virtual machine, without instrumenting the source code.

The availability of versioning systems and bug tracking systems, and of techniques to mine data from them (e.g., [59]) pose the basis for a third dimension in software analysis, namely the *Historical Analysis* of data extracted from software repositories [13]. Change *diffs* between file revisions, change coupling relationships between files being co-changed during the same time window, relationships between bug reports and code repositories, and the change rationale as documented in problem fixing reports and in CVS messages, are valuable sources of information complementary to static and dynamic analysis.

Different kinds of analyzers can be combined to build *Hybrid Analyzers*. This can have the purpose of combining analyses performed on different kinds of artifacts, of analyzing software systems developed using multiple technologies and programming languages [42]. Above all, hybrid analyzers can combine static and dynamic analysis [25, 52], since dynamic analysis can be incomplete, because it depends on program inputs, while static analysis can be imprecise, e.g., not able to fully deal with aspects such as pointer resolution.

5 Reverse Engineered Software Views

As explained in Section 2, reverse engineering aims at producing software views by means of (i) extracting information from the available software artifacts and (ii) performing a series of abstractions. Concepts relevant to software views are shown in Figure 5. A *Software View* is a representation of *Software Artifacts* and *Artifact Relationships*, able to satisfy one or more *Understanding Goals* expressed by a *Software Engineer*. Depending on the artifacts represented, a software view can be, for example (the list is not exhaustive):

- an *Architectural View*, such as those reconstructed by the *Bauhaus* tool [34] and the *Rigi* tool [57];
- a *Code View*, e.g., aimed at visualizing clones [37], slicers and data flow analyzers such as *CodeSurfer* [28];
- a *Metric View*, representing software artifact properties, metrics in particular. Examples include polymetric views [36], able to represent multiple metrics of a software artifact (e.g., number of methods, attributes and LOC of a class) by using rectangles width, height and color;
- an *Historical View*, for example the *animated storyboards* proposed by Beyer and Hassan [5] that, using a sequence of animated panels, visualize the changes occurring in a software system.

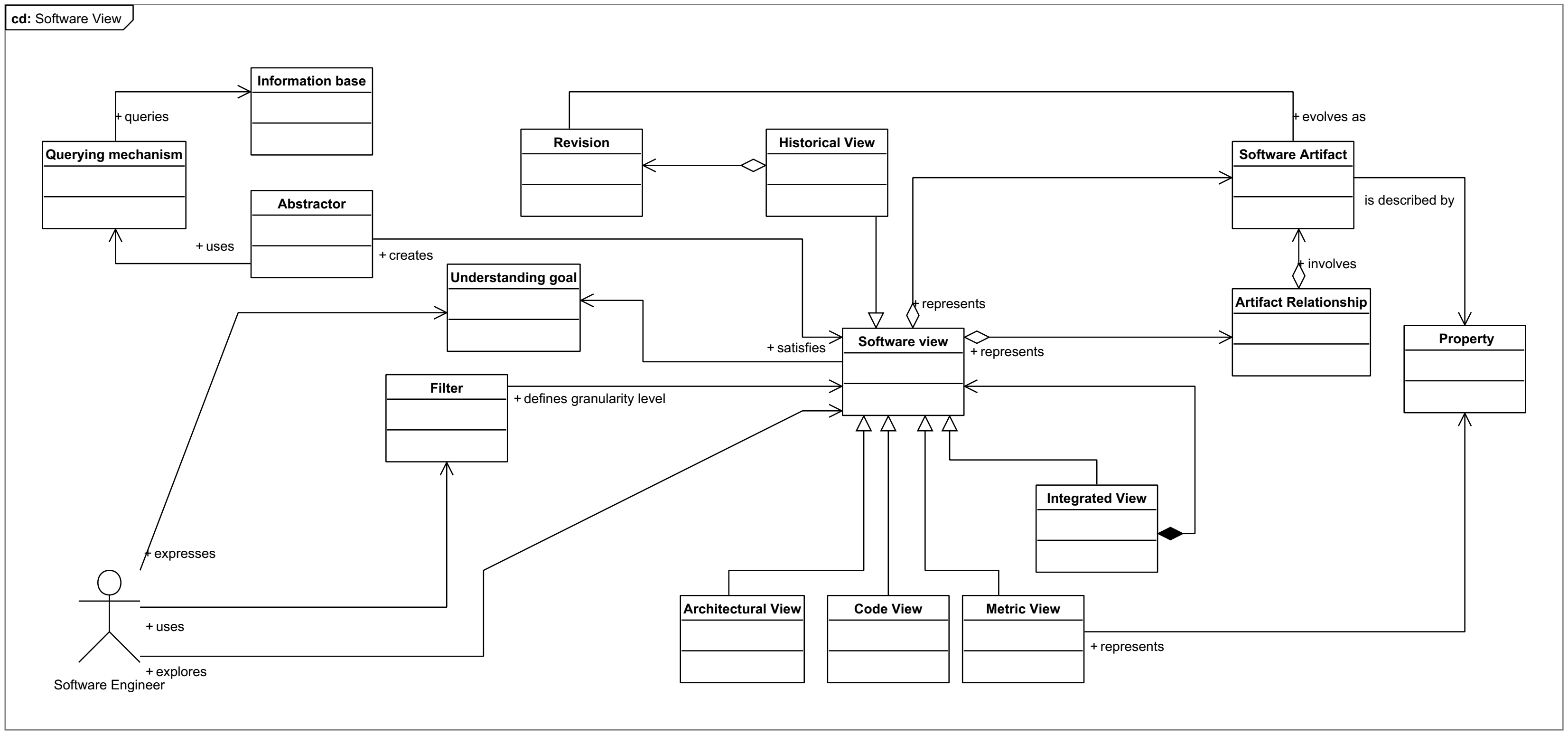

Figure 5. Views

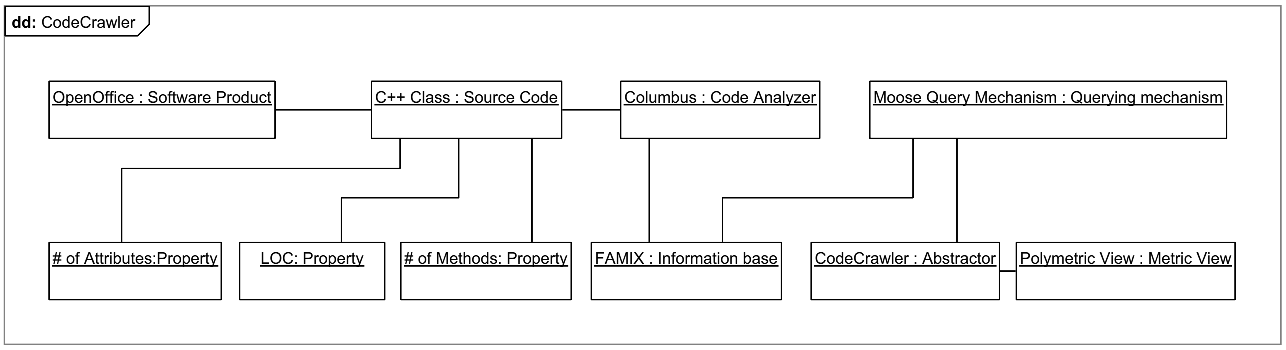

Figure 6. Conceptual model instance: producing polymetric views with CodeCrawler [36]

An important feature software views should provide is the capability to represent a software system at different levels of details, either providing a view in-the-large of the entire system, or focusing on fine-grained details. This can be made through *Filters*. Furthermore, it is often desirable for a reverse engineering tool to provide *Integrated Views*, able to visualize software artifacts at different levels of abstractions (e.g., from analysis models to source code) and to navigate among them.

To create a software view, the *Abstractor* queries the *Information Base* using a *Querying Mechanism*. An example of querying mechanism for information bases is *Grok* [29], a notation for manipulating binary relations using the Tarski's relational algebra [53]. *Grok* comprises operators for manipulating sets (e.g., union, intersection, difference) or relations (Cartesian product, transitive closure), and can be used to query information bases represented using the *TA* language. In the paper [29], Holt represents architectures as nested sets of box and arrow diagrams, and shows how the Tarski's relational algebra, and *Grok* in particular, can be used to produce architectural views of large and complex programs. Several systems implement Turing complete languages to query the information base and build software views. For example, the language implemented in the *Software Refinery* toolkit [39] supports different paradigms, including the procedural, object-oriented, and declarative paradigms, and the latter in both logic and transformational style. Logic programming, and in particular the Prolog programming language, has also been advocated as a means for querying an information base [9]. The main reasons for using Prolog are the declarative programming style, which is well suited to express information needs, and the metaprogramming capabilities, that ease the construction of abstraction facilities that accept user feedback to interactively improve the views.

The *GuPRO* tool [23] uses the GReQL (Graph Repository Query Language), which is essentially a declarative language for querying graph structures. Another example of querying mechanism is *CrocoPat* [6], which uses relational calculus to query information RSF information bases. Systems such as DMS [4] or TXL [18] essentially perform abstractions by performing pattern matching over ASTs. Finally, the ASTs produced by *SrcML* [17]—being represented in XML—do not need a customized query language, but can be queried using the XQuery language.

6 Model Instance: Moose and CodeCrawler

With the purpose of showing how the conceptual model proposed in this paper can be applied to describe reverse engineering processes and tools, this section instantiates it on the *Moose* environment and the *CodeCrawler tool*. We assume that a software engineer wants to produce a view of some metrics—number of methods, number of attributes, and LOC—of C++ classes composing a software system. Figure 6 shows an object model obtained by instantiating our conceptual model to the production of polymetric views [36].

First, classes are parsed by means of an external analyzer—*Columbus* [27] in particular— and then facts are stored in a information base using the *FAMIX* [22] metamodel. The information base is then queried using the *Moose* querying mechanism by *CodeCrawler*, which produces the polymetric views.

7 Concluding remarks

Reverse engineering encompasses a wide array of tasks related to comprehending and enhancing software. After the seminal paper by Chikofsky and Cross [15], which created a taxonomy for reverse engineering and related terms, in this paper we have offered a conceptual model for key reverse engineering concepts and their relationships. The model is specified using UML class diagrams. To ease understanding, we have instantiated it on a simple scenario. Whilst the current model is by no means exhaustive, it is compact and extensible, which enables the possibility of adding new concepts and relationships while minimizing the chance of introducing redundancies and inconsistencies.

We believe that the model can provide a means to systematize the state of the art and to establish a coherent framework to exchange, and compare, research work and achievements. In the future, we plan to use our model to drive a systematic review of the literature and a classification of research results and tools.

8 Acknowledgments

Massimiliano Di Penta is partially supported by the project METAMORPHOS (MEthods and Tools for migrAting software systeMs towards web and service Oriented aRchitectures: exPerimental evaluation, usability, and tecHnOlogy tranSfer), funded by MiUR (Ministero dell'Università e della Ricerca) under grant PRIN2006-2006098097. The authors thank Luigi Cerulo for his precious comments on early drafts of this paper.

References

[1] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Trans. Software Eng.*, 28(10):970–983, 2002.

[2] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, pages 361–370, 2006.

[3] L. Aversano, G. Canfora, A. Cimitile, and A. De Lucia. Migrating legacy systems to the web: An experience report. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, pages 148–157, 2001.

[4] I. D. Baxter, C. Pidgeon, and M. Mehlich. DMS: program transformations for practical scalable software evolution. In *26th International Conference on Software Engineering (ICSE 2004), 23-28 May 2004, Edinburgh, United Kingdom*, pages 625–634, 2004.

[5] D. Beyer and A. E. Hassan. Animated visualization of software history using evolution storyboards. In *13th Working Conference on Reverse Engineering (WCRE 2006), 23-27 October 2006, Benevento, Italy*, pages 199–210, 2006.

[6] D. Beyer and C. Lewerentz. CrocoPat: Efficient pattern analysis in object-oriented programs. In *11th International Workshop on Program Comprehension (IWPC 2003), May 10-11, 2003, Portland, Oregon, USA*, pages 294–295, 2003.

[7] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.16: components for transformation systems. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, 2006, Charleston, South Carolina, USA, January 9-10, 2006*, pages 95–99, 2006.

[8] M. L. Brodie and M. Stonebraker. *Migrating Legacy System*. Morgan Kaufmann, San Mateo, Calif.,USA, 1995.

[9] G. Canfora, A. Cimitile, and U. de Carlini. A logic-based approach to reverse engineering tools production. *IEEE Trans. Software Eng.*, 18(12):1053–1064, 1992.

[10] G. Canfora, A. Cimitile, A. De Lucia, and G. A. Di Lucca. Decomposing legacy programs: a first step towards migrating to client-server platforms. *Journal of Systems and Software*, 54(2):99–110, 2000.

[11] G. Canfora and M. Di Penta. New frontiers of reverse engineering. In *International Conference on Software Engineering (ICSE 2007) - Future of Software Engineering Track (FOSE)*, pages 326–341, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[12] G. Canfora, A. R. Fasolino, G. Frattolillo, and P. Tramontana. A wrapping approach for migrating legacy system interactive functionalities to service oriented architectures. *Journal of Systems and Software*, 81(4):463–480, 2008.

[13] L. Cerulo. *On the Use of Process Trails to Understand Software Development*. PhD thesis, RCOST - University of Sannio, Italy, 2006.

[14] Y.-F. Chen, M. Y. Nishimoto, and C. V. Ramamoorthy. The C information abstraction system. *IEEE Trans. Software Eng.*, 16(3):325–334, 1990.

[15] E. Chikofsky and J. I. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, Jan 1990.

[16] C. Cifuentes and K. J. Gough. Decompilation of binary programs. *Softw., Pract. Exper.*, 25(7):811–829, 1995.

[17] M. L. Collard, H. H. Kagdi, and J. I. Maletic. An xml-based lightweight c++ fact extractor. In *11th International Workshop on Program Comprehension (IWPC 2003), May 10-11, 2003, Portland, Oregon, USA*, pages 134–143, 2003.

[18] J. R. Cordy, T. R. Dean, A. J. Malton, and K. A. Schneider. Source transformation in software engineering using the TXL transformation system. *Information & Software Technology*, 44(13):827–837, 2002.

[19] T. R. Dean, A. J. Malton, and R. C. Holt. Union schemas as a basis for a C++ extractor. In *Proceedings of the Working Conference on Reverse Engineering*, pages 59–, 2001.

[20] M. Di Penta, M. Neteler, G. Antoniol, and E. Merlo. A language-independent software renovation framework. *Journal of Systems and Software*, 77(3):225–240, 2005.

[21] E. Duala-Ekoko and M. P. Robillard. Tracking code clones in evolving software. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, pages 158–167, 2007.

[22] S. Ducasse, M. Lanza, and S. Tichelaar. Moose: an extensible language-independent environment for reengineering object-oriented systems. In *Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000)*, 2000.

[23] J. Ebert, B. Kullbach, V. Riediger, and A. Winter. GUPRO - Generic Understanding of Programs. *Electr. Notes Theor. Comput. Sci.*, 72(2), 2002.

[24] E. Eilam. *Reversing: Secrets of Reverse Engineering*. Wiley, April 2005.

[25] M. D. Ernst. Static and dynamic analysis: synergy and duality. In *ICSE Workshop on Dynamic Analysis (WODA)*, Portland, Oregon, USA, May 2003.

[26] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, 2001.

[27] R. Ferenc, Á. Beszédes, M. Tarkiainen, and T. Gyimóthy. Columbus - reverse engineering tool and schema for C++. In *Proceedings of the International Conference on Software Maintenance*, pages 172–181. IEEE Computer Society, 2002.

[28] Grammatech Inc. The CodeSurfer slicing system, 2002.

[29] R. C. Holt. Structural manipulations of software architecture using Tarski relational algebra. In *Proceedings of the Working Conference on Reverse Engineering*, pages 210–219, 1998.

[30] R. C. Holt, A. Schürr, S. E. Sim, and A. Winter. GXL: A graph-based standard exchange format for reengineering. *Sci. Comput. Program.*, 60(2):149–170, 2006.

[31] J. Huffman Hayes, A. Dekhtyar, and S. K. Sundaram. Advancing candidate link generation for requirements tracing: The study of methods. *IEEE Trans. Software Eng.*, 32(1):4–19, 2006.

[32] IEEE. *std 1219: Standard for Software maintenance*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1998.

[33] R. Kazman, S. S. Woods, and S. J. Carrière. Requirements for integrating software architecture and reengineering models: Corum II. In *Proceedings of the Working Conference on Reverse Engineering*, pages 154–163, 1998.

[34] R. Koschke. *Atomic Architectural Component Recovery for Program Understanding and Evolution*. PhD thesis, Univ. of Stuttgart, Germany, 2000.

[35] R. Lämmel and C. Verhoef. Cracking the 500-language problem. *IEEE Software*, 18(6):78–88, 2001.

[36] M. Lanza and S. Ducasse. Polymetric views - a lightweight visual approach to reverse engineering. *IEEE Trans. Software Eng.*, 29(9):782–795, 2003.

[37] S. Livieri, Y. Higo, M. Matsushita, and K. Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed CCFinder: D-CCFinder. In *ICSE*, pages 106–115, 2007.

[38] A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using Latent Semantic Indexing. In *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003), May 3-10, 2003, Portland, Oregon, USA*, pages 125–137, 2003.

[39] L. Markosian, P. Newcomb, R. Brand, S. Burson, and T. Kitzmiller. Using an enabling technology to reengineer legacy systems. *Commun. ACM*, 37(5):58–70, 1994.

[40] E. Merlo, P.-Y. Gagné, J.-F. Girard, K. Kontogiannis, L. J. Hendren, P. Panangaden, and R. de Mori. Reengineering user interfaces. *IEEE Software*, 12(1):64–73, 1995.

[41] B. S. Mitchell and S. Mancoridis. On the automatic modularization of software systems using the Bunch Tool. *IEEE Trans. Software Eng.*, 32(3):193–208, 2006.

[42] D. L. Moise, K. Wong, H. J. Hoover, and D. Hou. Reverse engineering scripting language extensions. In *Proceedings of the 14th International Conference on Program Comprehension (ICPC 2006), June 2006, Athens, Greece*, pages 295–306, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[43] L. Moonen. Generating robust parsers using island grammars. In *Proceedings of the Working Conference on Reverse Engineering*, pages 13–22, 2001.

[44] M. Moore. *User Interface Reengineering*. PhD thesis, Georgia Institute of Technology, USA, 1998.

[45] H. A. Müller, J. H. Jahnke, D. B. Smith, M.-A. D. Storey, S. R. Tilley, and K. Wong. Reverse engineering: a roadmap. In *ICSE - Future of SE Track*, pages 47–60, 2000.

[46] G. C. Murphy. Houston: We are in overload. In *"Proceedings of IEEE International Conference on Software Maintenance (ICSM)*, Oct. 2007.

[47] M. J. Rekoff. On reverse engineering. *IEEE Trans. Systems, Man, and Cybernetics*, pages 244–252, March-April 1985.

[48] P. Resnick and H. R. Varian. Recommender systems. *Special Issue of the Commun. ACM*, 40(3):56–89, 1997.

[49] M. Siff and T. W. Reps. Identifying modules via concept analysis. *IEEE Trans. Software Eng.*, 25(6):749–768, 1999.

[50] H. M. Sneed. Encapsulation of legacy software: A technique for reusing legacy software components. *Ann. Software Eng.*, 9:293–313, 2000.

[51] T. A. Standish. An essay on software reuse. *IEEE Trans. Software Eng.*, 10(5):494–497, 1984.

[52] T. Systä. *Static and Dynamic Reverse Engineering Techniques for Java Software Systems*. PhD thesis, Univ. of Tampere, Finland, 2000.

[53] A. Tarski. On the calculus of relations. *J., Symbolic Logic*, 6(3):73–89, 1941.

[54] P. Tonella and A. Potrich. *Reverse Engineering of Object Oriented Code*. Springer-Verlag, Berlin, Heidelberg, New York, 2005.

[55] V. Tzerpos and R. C. Holt. Software botryology: Automatic clustering of software systems. In *DEXA Workshop*, pages 811–818. IEEE Computer Society Press, Los Alamitos, CA, USA, 1998.

[56] A. Winter, B. Kullbach, and V. Riediger. An overview of the GXL graph exchange language. In *Proc. of the Software Visualisation International Seminar, LNCS 2269*, pages 324–336, Dagstuhl Castle, Germany, May 2002. Springer-Verlag.

[57] K. Wong, S. Tilley, H. A. Muller, and M. D. Storey. Structural redocumentation: A case study. *IEEE Software*, pages 46–54, Jan 1995.

[58] Y. Yu, J. Mylopoulos, Y. Wang, S. Liaskos, A. Lapouchnian, Y. Zou, M. Littou, and J. C. S. P. Leite. RETR: reverse engineering to requirements. In *12th Working Conference on Reverse Engineering (WCRE 2005), 7-11 November 2005, Pittsburgh, PA, USA*, page 234, 2005.

[59] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 563–572. IEEE Computer Society, 2004.