2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR) | 979-8-3503-1184-6/23/$31.00 ©2023 IEEE | DOI: 10.1109/MSR59073.2023.00029

Enabling Analysis and Reasoning on Software Systems through Knowledge Graph Representation

Satrio Adi Rukmono
Mathematics and Computer Science
Eindhoven University of Technology
Eindhoven, The Netherlands
s.a.rukmono@tue.nl

Michel R.V. Chaudron
Mathematics and Computer Science
Eindhoven University of Technology
Eindhoven, The Netherlands
m.r.v.chaudron@tue.nl

***Abstract*—This work presents a knowledge-representation-based approach for analysing software systems. Its main components are: a generic and extensible knowledge model, and a knowledge extractor tool that generates instance-level knowledge graphs from software repositories (currently Java). Our knowledge model can be used as a shared data-model in a software analysis pipeline. We illustrate the potential uses of our knowledge representation by performing experimental architecture recovery and identifying design pattern instance. We intend to use our ontology and extraction tool as a partial foundation for automated reasoning on software systems.**

***Index Terms*—knowledge graph, software knowledge, software ontology, object-oriented, software analysis**

I. Introduction

As software is becoming more and more complex, we see a promising future for automated intelligent assistants for supporting developers of software systems. With the advent of large corpora of source code, analyses of software is becoming driven by heuristics rather than by strict rules. Our interest is in being able to analyse software both at a high level (software architecture/design) and at the implementation level.

As a stepping stone toward that vision, we need a good representation of software knowledge. We aim for our representation and extraction tools to serve as building blocks in software analysis infrastructures [1] such as SAIN [2]. In other words, we expect large-scale (semi-)automated analysis pipelines to be built on top of the knowledge-representation that is produced by our tools. For representing software for reasoning about its architecture/design, we identify the following requirements.

R1: Abstract & Concrete. We want our knowledge representation to support an abstract view that is free from implementation details so as to enable architecture-level analysis. This means the representation does not have to be back-convertible into source code. For example, our representation should not differentiate whether a data-structure is a `struct`, `class`, or `record`. At the same time, we want to be able to support relating high level views to concrete implementation level.

R2: Rich in relational information. Dependency graphs are commonly used in software analyses. These are often based on method/function calls. However, there are many other types of relationships between code fragments other than invocations. We want our knowledge representation to capture, for example, specialization and composition information.

R3: Language-agnostic. We want the knowledge representation not to be bound to a single programming language. We imagine the reasoning should be generic and apply to different implementation languages. However, for this work, we set our scope to include only class-based object-oriented software systems.

R4: Handles partial and evolving knowledge. From the models we find in practice, we know that knowledge about a system may be incomplete, yet still sufficient to enable basic types of analyses or reasoning. We want our representation to be able to capture such partial knowledge. This contrasts with classical modeling methods from the area of Formal Methods that require complete models of a system before any reasoning can be done.

R5: Supports multiple knowledge sources. Software knowledge may come from diverse sources such as source code, architecture documents, or UML models. We want to accumulate this different types of knowledge from different kinds of sources in a single representation.

R6: Enrichable. In addition to supporting different defined data sources, our knowledge representation should also be flexible enough for possible integration with emerging types of knowledge. For example, when classes are classified into role stereotypes [3], it should be possible to use this new information in an analysis pipeline.

We present our metamodel for describing software knowledge and an extractor implementation that works on Java projects, which we call javapers.

II. Related Work

There are several existing models and knowledge extraction tools that partially satisfy our requirements. In this section, we describe related work that inspired our requirements.

A. Software Metamodels

FAMIX is an abstract representation of source code that is generic and aims to describe systems written in multiple programming languages [4]. We largely adopt FAMIX concepts for our own knowledge representation, with some minor

2574-3864/23/$31.00 ©2023 IEEE
DOI 10.1109/MSR59073.2023.00029

refinements and extension. In particular, FAMIX family of metamodels do not take into account information on field access or method invocation that is not performed via a method. We add concepts concerning access and invocation that happen from an initializer (e.g., in Java) or within a class definition but outside any of its methods (e.g., in Ruby).

A graph-based metamodel introduced by Mens and Lanza [5] includes details like line number and method calls, but it does not have grouping information to accommodate (Java) packages or (C#) namespaces, for example.

B. Existing Tools

srcML [6] annotates code fragments into an XML file. It does well in tagging structural parts of code in a language-agnostic manner, but does not provide relational information. For example, it may tag that method M_1 of class C_1 invokes method M_2, but it does not resolve which class contains the definition of M_2.

CodeOntology is an RDF-based software knowledge representation [7]. It satisfies many of our requirements. However, it represents knowledge in high-fidelity in terms of language constructs. As a result, things that have similar abstract semantics would be represented differently if they are implemented in different language, complicating generic analysis on the representation.

Various work, notably in the area of software architecture recovery (SAR) [8], [9], describe "dependency extractors" to build dependency graph of a software system. As noted in our requirements, these are mostly limited to method calls.

DP-CORE [10] is a design pattern instance detector. In intermediate steps to detect patterns, it identifies similar kind of relationships to what we would like to have. However, it does not produce an externally usable representation of this information.

inspect4py [11] has a comparable goal of extracting knowledge from code. It extracts software understanding and code features from Python source code. Its direction differs to ours in terms of the type of information extracted (and thus, knowledge gained).

SAIN [2]: In contrast to the above, SAIN is not a single knowledge extraction tool. Rather, it is a platform for building advanced software analysis pipelines, where our tool could be one of its building blocks.

III. Knowledge Representation Design

We represent the knowledge we extract about software using knowledge graphs. The approach we take on knowledge representation is to split the knowledge into a generic part and a project-specific part. We capture generic knowledge as an ontology that we make manually. This ontology serves as a metamodel that describes a knowledge instance.

We opt to adopt labeled property graph as the representation format. We choose labeled property graph over RDF mainly because we find that some information inherently describes a particular concept rather than stands alone. An example is a class name, which inherently describes a class. In an RDF environment, both a class and a class name are first-order objects, resulting in a significantly larger knowledge graph that necessitates complex queries for analyses. Furthermore, with RDF, it is not possible to differentiate instances of relationships of the same type. Additionally, anecdotal evidences in an online discussion[1] suggest that the query language Cypher [12] for labeled property graph is perceived as simpler to understand than RDF's SPARQL [13].

We made a two-level ontology for different levels of abstractions. The first is a detailed one that includes the concepts of types, classes, fields, and methods as graph nodes. The second one works on a higher level of abstraction and only considers containers and types (including classes). Classes in both levels of abstraction represent the same concept, so they serve as a connecting point between the two levels. Thus, the separation into two levels helps in supporting different types of analyses while still enabling the analysis results to be correlated.

We separate our ontology into two levels of abstraction to lower cognitive load when reasoning on higher level purposes (practically: queries are simpler) while still providing means to perform detailed analyses. While our ontology describes code-level entities, it is designed to support architecture-level analyses. We have designed a set of extensions to this ontology for imbuing architectural knowledge into the knowledge base; however, this is not in the scope of this paper.

We depict our ontology in the form of class diagrams that act as graph types, guiding what kind of node labels may exist and what kind of edge labels can there be between two kinds of node labels in our knowledge graphs. Classes in the diagram represent node labels, while associations represent edge labels.

A. Detailed Graph Type

Fig. 1 shows our graph schema for the lower-level of abstraction, having more details compared to the higher-level one. In addition to associations, class fields in the diagram denote edges as well. The class field notation is used to avoid cluttering the diagram. Additional information, if and when necessary, can be added as properties to the graph nodes and labels. Examples of such information are class names, method signatures, and the scope of methods (i.e., static/class-methods as opposed to object-methods).

We define the node labels of the knowledge graph as follows.

Structure: a class-type, including what languages may call *structs*, *records*, *enumerations*, etc. (The definitions of and *Structure*'s relations to *Type* and *Primitive* are therefore trivial for people familiar with object orientation.)

Container: anything in the language/platform that contains *Structures* or allows organization of *Structures*, e.g., *package*, *directory*, *module*, or *namespace*. *Structure* extends *Container* because some programming languages allow nested classes.

Variable is self-explanatory; however, we point out that it includes what languages may call *field*, *attribute*, *property*, function/method *parameter*, etc.

[1] https://news.ycombinator.com/item?id=21004610, accessed 2023-01-25

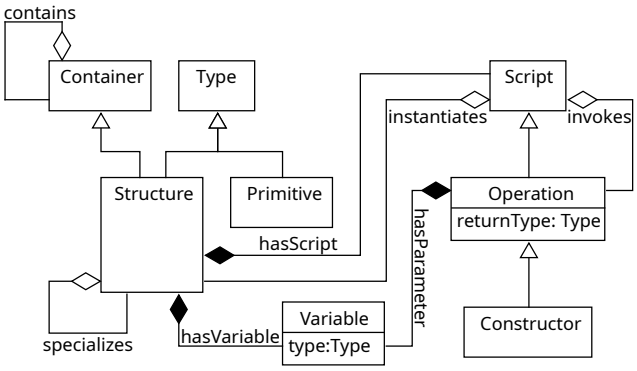

Fig. 1. Our lower-level (detailed) ontology for object-oriented system.

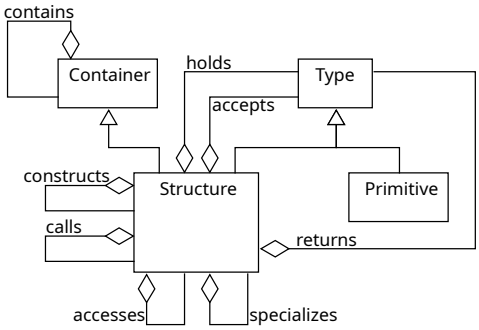

Fig. 2. Our higher-level (abstract) ontology.

Script: a code expression or a block of statements; lines of code that produce value or make effects.

Operation: a kind of *Script* that is identified with a signature (name and parameters) and return type.

Constructor: a kind of *Operation* whose sole purpose is the creation of a *Structure* instance. We consider a *Constructor*'s return type to be the *Structure* that it creates, even though Java considers constructors to have no return type.

We define the edge labels as follows:

Specializes is for inheritance, including Java's `extends` and `implements` keywords, *mixins*, etc.

Invokes denotes invocations. A *Script* that does not have a signature (i.e., not an *Operation*) cannot be referred to, and therefore *invoked* by, another *Script*. An example is field initializers.

Instantiates is a convenience edge that can be derived from other terms in the graph. $\boldsymbol{M}$ **instantiates** $\boldsymbol{S}$ when *Structure* S has a *Constructor* C that is invoked by *Script* M.

The other edge types *contains*, *type*, *returnType*, *hasVariable*, *hasScript*, and *hasParameter* should be self-explanatory.

As can be seen, our ontology focuses on (class-based) object-oriented systems, while still covering concepts from structured programming. It is not designed with functional programming in mind; however, if we consider a function to be a *Structure* containing a specific *Operation*, this ontology arguably supports *higher-order function*, which is an important concept in functional programming.

B. Abstract Graph Type

Fig. 2 shows our higher-level graph schema. This schema includes only *Containers* and *Types* as nodes, with relations between nodes derived from patterns of relations from the expanded model.

When a detailed graph is available, an abstracted version can be derived by applying the following definitions of edges and then removing *Variable* and *Script* nodes (and consequently their connected edges). $\boldsymbol{S}$ **holds** $\boldsymbol{T}$ when *Structure* S has a variable of *Type* T. $\boldsymbol{S}$ **accepts** $\boldsymbol{T}$ when *Structure* S has an operation that has a parameter of *Type* T. $\boldsymbol{S}$ **returns** $\boldsymbol{T}$ when *Structure* S has an operation with return *Type* T. $\boldsymbol{S}$ **constructs** $\boldsymbol{T}$ when *Structure* S has a script that instantiates *Structure* T. $\boldsymbol{S}$ **calls** $\boldsymbol{T}$ when *Structure* S has a script that calls an operation of *Structure* T. It is what is usually considered in a "dependency graph". $\boldsymbol{S}$ **accesses** $\boldsymbol{T}$ when *Structure* S has a script that directly access a field of *Structure* T.

IV. JAVAPERS

We developed a tool called javapers [14]. It takes a directory of Java source files (up to source level 16) and extracts knowledge graph instances that comply with our ontology. It supports extracting both the abstracted and detailed knowledge graph instances. It leverages the Spoon library [15].

Use the following command to run javapers, assuming that the binary archive is accessible from the working directory.

```
$ java -jar javapers.jar <args>
```

The arguments are described below.

`-i [path]` – path to the source directory of the Java project.
`-o [path]` – path to the directory where output file(s) will be created.
`-f [json|csv]` – the output format. Default value is `csv`.
`-n [name]` – output file basename. For example, when output type is `json`, the output file name will be `[name].json`.
`-c` – when this argument is provided, generate an abstracted knowledge graph. Otherwise, generate a detailed one.

The project is available on GitHub [14] as a Kotlin project using Maven as build system. It requires JDK 17 and Kotlin 1.8. Follow standard Maven procedures for building binaries.

Tool limitations: Generic type arguments and array types are "lifted". In effect, for example, a field of type `List<String>` and `String[]` are both treated as a field of type `String`. This should be taken into account when relevant while performing analyses on a knowledge graph produced using our tool.

V. PRELIMINARY EVALUATION

We demonstrate the knowledge graph instance extraction and two kinds of analysis on the graph: software architecture recovery and design pattern instance detection. We run javapers on JHotDraw version 5.1, a vector-drawing application and framework developed to showcase design pattern usages. The project consists of 154 Java classes. The resulting

knowledge graphs of JHotDraw are available along with the source code of javapers. A web-based interactive visualization of the knowledge representation is also available.[2]

Software architecture recovery: Graph community detection algorithms can be used for software architecture recovery, where communities of classes correspond to cohesive components. We experiment with the Leiden technique [16] on an abstract knowledge graph instance describing JHotDraw and identify 14 components that by manual inspection seem to contain semantically related classes. For example, one component contains application controller- and corresponding GUI-classes, while another component contains classes related to data structure and related persistence classes. We find that patterns in relationships correspond with architectural layers and functional decomposition. For instance, *constructs* relationships prominently come from a higher architectural layer while *calls* and *accepts* usually occur within a layer. However, this is only an initial experiment and requires further refinement and proper evaluation.

Design pattern detection: Existing techniques can be used to detect design patterns by analyzing patterns of relation types between classes [10], [17], [18]. We apply such analysis to our language-agnostic representation using Cypher queries. To illustrate an analysis on our knowledge graph, we start with ground truth information on a design pattern instance in JHotDraw from the P-MARt repository [19]. We select an interface `Connector` that acts as an abstract *strategy interface* in a Strategy pattern instance. We then use our interactive visualization tool to inspect the neighboring nodes connected with edge types relevant to a Strategy pattern, resulting in a sub-graph depicting this particular pattern instance (Fig. 3). The figure is stylized using familiar UML notations and a screencast describing the analysis process is linked from the visualization page.

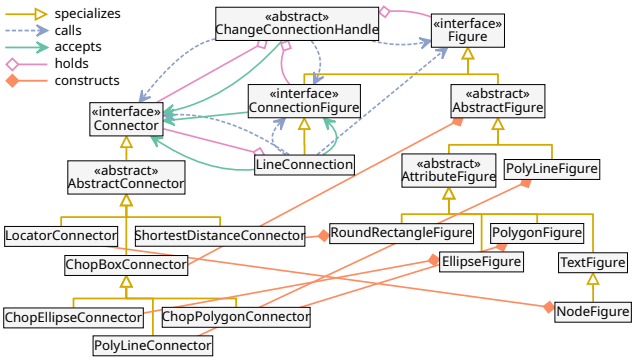

Fig. 3. A subset of the abstract knowledge graph instance extracted from JHotDraw, depicting classes involved in a strategy design pattern, presented in UML-like styling.

Classes `ChangeConnectionHandle`, `ConnectionFigure`, and `LineConnection` act as the context that accepts a reference to a `Connector` and calls its methods. `Figure` provides an interface that lets the strategy access its data. The concrete strategies (a cluster of `Connector` subclasses) are constructed depending on the concrete `Figure` involved (a cluster of `AbstractFigure` subclasses). We cross-check this finding with the ground truth and find that we discover 16 out of 19 classes that experts manually identified to belong in this pattern instance.

Based on the relation pattern in this instance, we build a Cypher query below that can be used as a starting point to look for instances of the Strategy pattern.

```
MATCH p=(c)-[:ACCEPTS]->(s)<-[:SPECIALIZES*1..]-
  (cs)<-[:CONSTRUCTS]-()-[:SPECIALIZES*]->(i)
  <-[:HOLDS*]-(c),
  q=(c)-[:CALLS]->(s),
  r=(c)-[:HOLDS]->(s)
WHERE (i)<-[:CALLS|HOLDS]-()<-[:SPECIALIZES*]-(cs)
RETURN p,q,r
```

This is a first step in an iterative process of querying the knowledge graph, inspecting the result, and refining the query.

VI. Future Work

Our tool is a first step towards knowledge-based software system analyses. We envision it and the accompanying visualization tool evolving into a platform that integrates multiple analysis and reasoning tools.

A logical next step is to build an interactive query environment to support graph analyses, such as detecting design pattern instances as described above. Our system produces various (UML-like) visualizations based on the knowledge-graph, with possible enrichments.

As we plan to explore heuristic- and machine learning-based analyses, we are looking into transforming our knowledge (sub)graphs into a vector representation for use in neural network models, similar to code2vec [20], [21] and related works [22], [23] but in a higher-level of system abstraction.

Currently, our implementation only supports Java, but we plan to add C++ support. Implementing an extractor supporting another language requires an AST walker and type resolver for that language. ClaiR [24] seems to cover this need for C++.

We exclude knowledge about external libraries to focus on the system under review, but plan to add this in the future.

VII. Conclusions

We present a system that combines ontologies and knowledge graphs to create a generic and extensible system for analysing and reasoning about software systems. It enables the use of knowledge-based techniques for integrating knowledge from diverse sources, and also enables applying knowledge-based reasoning techniques. We encourage the community to use our tool and build software-analysis tools on top of it.

Acknowledgment

Initial exploratory versions for javapers utilize Rascal [25] and Groove [26]. We would like to thank Jurgen Vinju (Rascal) and Arend Rensink (Groove) for the conversations that inspire the direction of our research.

[2]https://rsatrioadi.github.io/classviz/?p=jhotdraw_abstract

REFERENCES

[1] T. Ho-Quang, M. R. V. Chaudron, G. Robles, and G. B. Herwanto, "Towards an infrastructure for empirical research into software architecture: challenges and directions," in *2019 IEEE/ACM 2nd International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE)*. IEEE, 2019, pp. 34–41.

[2] J. Garcia, M. Mirakhorli, L. Xiao, Y. Zhao, I. Mujhid, K. Pham, A. Okutan, S. Malek, R. Kazman, Y. Cai *et al.*, "Constructing a shared infrastructure for software architecture analysis and maintenance," in *2021 IEEE 18th International Conference on Software Architecture (ICSA)*. IEEE, 2021, pp. 150–161. [Online]. Available: www.sain.info/

[3] T. Ho-Quang, A. Nurwidyantoro, S. A. Rukmono, M. R. V. Chaudron, F. Fröding, and D. N. Ngoc, "Role stereotypes in software designs and their evolution," *Journal of Systems and Software*, vol. 189, p. 111296, 2022.

[4] S. Ducasse, N. Anquetil, M. U. Bhatti, A. Cavalcante Hora, J. Laval, and T. Girba, "MSE and FAMIX 3.0: an interexchange format and source code model family," 2011.

[5] T. Mens and M. Lanza, "A graph-based metamodel for object-oriented software metrics," *Electronic Notes in Theoretical Computer Science*, vol. 72, no. 2, pp. 57–68, 2002.

[6] M. L. Collard, M. J. Decker, and J. I. Maletic, "srcML: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration," in *2013 IEEE International Conference on Software Maintenance*. IEEE, 2013, pp. 516–519.

[7] M. Atzeni and M. Atzori, "CodeOntology: RDF-ization of source code," in *International Semantic Web Conference*. Springer, 2017, pp. 20–28.

[8] V. Tzerpos and R. C. Holt, "ACDC: an algorithm for comprehension-driven clustering," in *Proceedings Seventh Working Conference on Reverse Engineering*. IEEE, 2000, pp. 258–267.

[9] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, and Y. Cai, "Enhancing architectural recovery using concerns," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 2011, pp. 552–555.

[10] T. Diamantopoulos, A. Noutsos, and A. Symeonidis, "DP-CORE: A design pattern detection tool for code reuse," *BMSD 2016*, p. 160, 2016.

[11] R. Filgueira and D. Garijo, "Inspect4py: A knowledge extraction framework for python code repositories," in *IEEE/ACM 19th International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022*. IEEE, 2022, pp. 232–236. [Online]. Available: https://dgarijo.com/papers/inspect4py_MSR2022.pdf

[12] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor, "Cypher: An evolving query language for property graphs," in *Proceedings of the 2018 international conference on management of data*, 2018, pp. 1433–1445.

[13] I. Kollia, B. Glimm, and I. Horrocks, "SPARQL query answering over OWL ontologies," in *The Semantic Web: Research and Applications: 8th Extended Semantic Web Conference, ESWC 2011, Heraklion, Crete, Greece, May 29-June 2, 2011, Proceedings, Part I 8*. Springer, 2011, pp. 382–396.

[14] S. A. Rukmono, "rsatrioadi/javapers: javapers 1.0," Jan. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.7568438

[15] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "Spoon: A library for implementing analyses and transformations of java source code," *Software: Practice and Experience*, vol. 46, pp. 1155–1179, 2015. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01078532/document

[16] V. A. Traag, L. Waltman, and N. J. Van Eck, "From Louvain to Leiden: guaranteeing well-connected communities," *Scientific reports*, vol. 9, no. 1, pp. 1–12, 2019.

[17] G. Rasool, "Customizable feature based design pattern recognition integrating multiple techniques," Ph.D. dissertation, Technische Universität Ilmenau, Germany, 2011.

[18] Z. Shahbazi, A. Rasoolzadegan, Z. Purfallah, and S. Jafari Horestani, "A new method for detecting various variants of GoF design patterns using conceptual signatures," *Software Quality Journal*, vol. 30, no. 3, pp. 651–686, 2022.

[19] Y.-G. Guéhéneuc, "P-MARt: Pattern-like micro architecture repository," *Proceedings of the 1st EuroPLoP Focus Group on pattern repositories*, pp. 1–3, 2007. [Online]. Available: https://www.ptidej.net/tools/designpatterns/

[20] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," *arXiv preprint arXiv:1808.01400*, 2018.

[21] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[22] H. Peng, L. Mou, G. Li, Y. Liu, L. Zhang, and Z. Jin, "Building program vector representations for deep learning," in *International conference on knowledge science, engineering and management*. Springer, 2015, pp. 547–553.

[23] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 783–794.

[24] R. Aarssen, "cwi-swat/clair: v0.1.0," Sep 2017.

[25] P. Klint, T. Van Der Storm, and J. Vinju, "Easy meta-programming with Rascal," *Generative and Transformational Techniques in Software Engineering III: International Summer School, GTTSE 2009, Braga, Portugal, July 6-11, 2009. Revised Papers*, pp. 222–289, 2011.

[26] A. H. Ghamarian, M. de Mol, A. Rensink, E. Zambon, and M. Zimakova, "Modelling and analysis using GROOVE," *International journal on software tools for technology transfer*, vol. 14, no. 1, pp. 15–40, 2012.